Description of Languages Based on Object-Oriented Meta-Modelling

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Inf. Markus Scheidgen
geboren am 8.4.1978 in Berlin

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter:

1. Prof. Dr. Joachim Fischer
2. Prof. Dr. Andreas Prinz
3. Prof. Dr. Andy Schürr

eingereicht am:	6. Oktober 2008
Tag der mündlichen Prüfung:	6. Februar 2009

Abstract

In this thesis, I look into object-oriented meta-modelling and how it can be used to describe computer languages. Thereby, I do not only focus on describing languages, but also on utilising the language descriptions to automatically create language tools from language descriptions. I use the notion of meta-languages and meta-tools. Meta-languages are used to describe certain language aspects, such as notation or semantics, and meta-tools are used to create language tools, such as editors or interpreters, from corresponding descriptions. This combination of describing and automated development of tools is known as domain specific modelling (DSM).

I use DSM based on object-oriented meta-modelling to describe all important aspects of executable computer languages. I look into existing meta-languages and meta-tools for describing language utterances, their concrete representation, and semantics. Furthermore, I develop a new platform to define languages based on the CMOF-model of the OMG MOF 2.x recommendations. I develop a meta-language and meta-tool for textual language notations. Finally, I develop a new graphical meta-language and meta-tool for describing the operational semantics of computer languages.

To prove the applicability of the presented techniques, I take SDL, the Specification and Description Language, as an archetype for textually notated languages with executable instances. For this archetype, I show that the presented meta-languages and meta-tools allow to describe such computer languages and allow to automatically create tools for those languages.

Zusammenfassung

In dieser Dissertation, schaue ich auf objekt-orientierte Metamodellierung und wie sie verwendet werden kann, um Computersprachen zu beschreiben. Dabei, fokussiere ich mich nicht nur auf die Beschreibung von Sprachen, sondern auch auf die Verwendung von Sprachbeschreibungen zur automatischen Erzeugung von Sprachwerkzeugen aus Sprachbeschreibungen. Ich nutze die Idee von Metasprachen und Metawerkzeugen. Metasprachen werden verwendet um bestimmte Sprachaspekte, wie Notationen und Semantiken, zu beschreiben, und Metawerkzeuge werden verwendet um Sprachwerkzeuge wie Editoren und Interpreter aus entsprechenden Beschreibungen zu erzeugen. Diese Kombination von Beschreibung und automatischer Entwicklung von Werkzeugen ist als Domänenspezifische Modellierung (DSM) bekannt.

Ich verwende DSM basierend auf objekt-orientierter Metamodellierung zur Beschreibung der wichtigen Aspekte ausführbarer Computersprachen. Ich untersuche existierende Metasprachen und Metawerkzeuge für die Beschreibung von Sprachvorkommen, ihrer konkreten Repräsentation und Semantik. Weiter, entwickle ich eine neue Plattform zur Beschreibung von Sprachen basierend auf dem CMOF-Modell der OMG MOF 2.x Empfehlungen. Ich entwickle eine Metasprache und Metawerkzeug für textuelle Notationen. Schlussendlich, entwickle ich eine graphische Metasprache und Metawerkzeug zur Beschreibung von operationaler Semantik von Computersprachen.

Um die Anwendbarkeit der vorgestellten Techniken zu prüfen, nehme ich SDL, die Specification and Description Language, als einen Archetypen für textuell notierte Sprachen mit ausführbaren Instanzen. Für diesen Archetyp zeige ich, dass die präsentierten Metasprachen und Metawerkzeuge es erlauben solche Computersprachen zu beschreiben und automatisch Werkzeuge für diese Sprachen zu erzeugen.

Contents

Acknowledgements

Although, I was told that writing a thesis is mainly a lone warrior's job, there were many people that supported me in doing all the research for this thesis as well as in writing this document itself. First of all, this work was only possible due to founding by the Deutsche Forschungs Gemeinschaft and the Land Berlin. I want to thank my thesis supervisor Prof. Dr. Joachim Fischer for giving me all the freedom and trust that allowed me to do that work on my own authority and with the needed independence that I think is necessary to create truly original scientific contributions. Furthermore, I have to thank him for his generous attitude in helping me financing my research, visiting conferences, and visiting fellow researches. I want to thank Prof. Dr. Andreas Prinz and Prof. Dr. Andy Schürr for discussing my ideas, referencing me towards existing related work, and showing me the parts of my thesis that needed improvement. I know that this kinda support from Professors is something that can regrettably not be taken for granted: thank you. I want to thank all my colleagues that I wrote conference contributions with. The discussions and exchange of ideas during these collaborations helped enormously to form the foundation of ideas for this thesis. Further thanks goes to Falko Theisselmann, Dirk Fahland, and Daniel Sadilek for taking the risk in using my tools for their own work. Their feedback and contributions helped notably to enhance my tools. I want to thank Daniel Sadilek, Guido Wachsmuth, and Stephan Weißleder, who where always available for technical or scientific discussions, or even just to help me to distract myself from too much work. Finally, I thank my dear friend Tolja for reminding me relentlessly that in the end this thing is just a doctoral thesis.

Preface

A few years ago, I was developing software for tele-communication service switches. The system basically was a rigidly model-driven engineered protocol stack, run on a complex middle-ware tailored for highly fail-safe systems. During this work, I was confronted with a very complex formal system design language, called Specification and Design Language (SDL). The software, I was working on, was written in such a way that it used almost every single language construct of SDL. At that time, I had to look up details about syntax and semantics of SDL on a daily basis. My document of reference was the SDL standard, a roughly 200 page word document containing two grammars, a few logical expressions, and (at least for me) confusing English text. In the end, I had to conclude on many occasions that not the standard determined the final behaviour of an SDL specification, but the used tools. This was either because the specification was ambiguous or the tool vendor did not implement the language as specified. In the end, not the standard was the final instance, it was the tool vendor of choice.

Purely incidentally, the tool vendor in my case was also the research group that I was about to write my diploma thesis with. The specific topic was on creating a meta-model for SDL, but the idea behind it was how to close the gap between language specification and tool. For almost all languages of that time, there was a specification and there was one or more tool implementations, but there was no hard connection between those two artefacts. Conclusively, not every tool implemented a language specification fully, correctly, or precisely. Even though there was a lot of technology that would allow to automatically derive language tools from language specifications, this was seldom used.

With UML-based software modelling and the model-driven software development movement, there was a paradigm shift in describing a language and implementing language tools from such language descriptions. Metamodelling based on UML class diagrams uses a software modelling technique to model (i.e. describe or specify) languages. The software and languages used to develop the software are described with the same means. Together

with combining different software models written in different languages in a model-driven development architecture, the borders between language and software began to melt, in the same way that the borders between language model and language tools became thinner. What started very innocently with the UML meta-model and model-driven architecture, goes now as far as the idea of language oriented programming, which completely integrates the process of developing software and the language used to develop the software.

However, using UML class diagrams to describe the constructs of a language was just a first step towards model-driven language development. To fully describe a language as a set of software models for language tools and actually automatically implement language tools from those models, several techniques were missing. At the time I started work on this thesis in 2004, one could conveniently describe the structure of a language and generate a programming library that allows to handle language utterance programmatically. This is a suitable backbone for language tools, but there is more to a complete set of language tools. For SDL as example, I wanted to have at least an editor and an SDL interpreter. This was the concrete goal, but in general I wanted to generalise things: how can editors be described, how can simulators be described, can this be done independently of the concrete language? During the course of my work, I did not simply build SDL tools, instead I jumped onto the popular domain specific modelling trend and created small description languages for certain language aspects and language tools. With these languages I could describe the SDL language, its notation, its semantics and generate the wanted tools automatically.

The final goal is a set of description languages that allows to write a language specification which can be used to automatically create language tools. This is a specification that is readable by human language users and at the same time determines the behaviour of language tools. In this thesis, take first steps towards this goal.

Chapter 1

Introduction

This thesis is about describing computer languages; it is about languages for describing languages, and it is about automated language tool development. Based on the assumption that describing a language to possible human users is not the only purpose of a language description, I want to show that language descriptions can also be used to automatically create tools for the described language. This subject is motivated by drastic changes in the character and use of computer languages: mathematical formal language descriptions and manual language tool development strategies, as developed for the early programming languages, are not adequate for today's quickly changing highly integrated programming, modelling, and domain specific languages. In this dissertation, I examine object-oriented meta-modelling, as a basis for language description techniques, which promise to describe languages in a human comprehensible and yet machine understandable way.

Before I can give an introduction and motivation, I need to create a framework of definitions that allows to clearly discuss the subject. With these definitions, I briefly introduce existing language description methods. Then, I introduce object-oriented meta-modelling, which I will examine as a language description methodology in this thesis. Following is a list of requirements for description techniques. Afterwards, I look at how software can be developed automatically based on descriptions and how this can be applied to the development of language tools. Finally, the introduction is closed with the motivation, aim, hypothesis, contributions and outline.

1.1 (Computer) Languages

Languages are means to convey information to something. *Computer languages are all those languages that are used to convey information to a com-*

puter. Not surprisingly, this intuitive definition describes computer languages as a means for communication. Because the receiving communication partners is a computer system, each utterance of a computer language has to be computer processable. Therefore, each language instance needs to have well defined structure and meaning. The receiving computer system needs to be programmed with means to analyse the structure of a language instance and comprehend its meaning.

As long as there is no confusion with another kind of language, I will refer to computer languages simply as languages. In the following sections, I define what languages and language utterances are and what structure (syntax), representation, and meaning of language utterances are.

1.1.1 Languages

As in formal language theory, a *language*[1] is a set of language utterances. Because this set forms a class or group of elements with common characteristics (they belond to the same language), I call these utterances *language instances*:

Definition 1 (language instance, language) *A* language instance *is an object with well defined structure and meaning, and a* language *is a set of language instances.*

I introduce the concept of *language description* as a means to determine what the instances of a language are. You will later see that there are several concrete methods, which realise this concept, e.g. meta-modelling and grammars.

Definition 2 (language description) *A* language description *is a finite system of rules that describes what are the instances of the described language.*

1.1.2 Syntax

A language instance is not a monolithic piece, it has a structure and is constructed from smaller parts. These parts are called *language constructs.*

[1] The definition and use of the term language in computer science literature is ambiguous and confusing. Formal language theory most often uses language as a set of words over an alphabet [66]. This is consisted to my definition, since words over an alphabet can be used as language instances. In the broader context of programming or modelling languages, languages are often considered a structure consisting, among others, of syntax and semantics [42]. Some authors use the term formalism (see 2.2.1) and language as synonyms (in [42] diagram languages are compared to visual formalisms), other use the terms formalism and language as in this theses [124].

Definition 3 (language construct) Language constructs *are the building blocks for language instances. Language constructs are defined by* construct definitions*. Each occurrence of a language construct within a language instance is a* construct instance*. The definition of a construct determines what the instances of this construct are.*

A construct definition can be related to other construct definitions. Construct instances can be connected to other construct instances according to the relations between the corresponding construct definitions.

A language instance is built from language constructs instances: a language instance comprises (connected) construct instances that form a structure. This structure is the *syntax*[2] of the language instance.

Definition 4 (syntax) *The* syntax *of a language instance is the structure formed by the language construct instances that constitute the language instance. The* syntax description *of a language is a collection of language construct definitions that describes the structures of possible language instances.*

The concepts syntax and language instance are closely related: syntax means the structure of an instance, language instance means the instance itself. In some cases both terms refer to the same thing. Take the language description method context-free grammars for example. A language instance has a concrete form, it is a string of terminal symbols; the syntax of this string is a tree-structure that reflects how this string was generated from grammar rules. Language instance (string) and syntax (tree-structure) are two different things. In other language description methods, such as MOF-like meta-modelling, a language instance does not have a concrete form, it is a structure, and this structure is also the syntax of the language instance. In MOF-like meta-modelling, concrete form (representation) is given to language instances by the use of notations, which are separately defined from languages (see the next section on representation and semantics).[3]

[2] In its original linguistic sense the term syntax refers to a set of rules, which describe the structure of language phrases and sentences (in my sense, the instances of languages) [35]. It determines how complex sentences can be built from simpler sentences. In formal language theory, syntax analysis constructs the sentences of a language by rules over an *alphabet* using grammars [66, 44]. To apply the terms syntax and languages more generally, e.g. to graphical languages, I omit the notions of sentences and alphabets and talk about language instances in general.

[3] The terms *concrete syntax* and *abstract syntax* are sometimes used to distinguish between concrete language instances and its (abstract) syntax when using context-free grammars for example. Or, in the case of language description methods that do not allow to define concrete language instances, *concrete syntax* and *abstract syntax* are used to distinguish between concrete representation of a language instance and the (abstract)

Similar to the terms *language instance* and *syntax*, the terms *language description* and *syntax description* seem to be synonyms, but *syntax description* is a weaker concept. A language description can consist of more than a syntax description. A language description can contain rules that further restrict the set of *possible* language instances. A syntax description provides construct definition, which determine how construct instances form the structures of language instances. Other rules of a language description can describe which of these structures are not valid instances of the language. These kind of rules or sometimes referred to as *constraints* or as *static semantics*. Based on its wrong reference towards *semantics*, the term *static semantics* is misleading and I will use *constraints* instead.

1.1.3 Representation and Semantics

A language instance has meaning; this is the information it conveys. A language allows to convey information from a certain *semantic domain*. Therefore, all instances of a language take their meaning from the same semantic domain. This domain determines the possible meanings of the instances of a language. A mapping between language and semantic domain determines, which language instance has what meaning. This mapping determines the language semantics.

Definition 5 (semantics) *A language instance has meaning (e.g. a statement, expression, command, program, software model, formula, etc.). This meaning is the* semantics *of a language instance. A* semantic domain *is a set of elements that comprise the meaning of the instances of a language. A* semantic mapping *is a relation between a language and a semantic domain. A semantic mapping and corresponding semantic domain together are called a* language semantics.

Multiple semantic mappings and multiple semantic domains can be used for the same language; i.e. a language can have different language semantics. In later chapters, characteristics for semantic mappings are discussed; for now, I only demand that a semantic mapping is left-total, i.e. each language

language instance itself or the (abstract) syntax thereof. These two terms are also used to refer to the description of (abstract) syntax and (concrete) representations. Especially for textual languages, these two terms are also used to distinguish between language instances constructed from non-terminals as they can be used in a text-file (concrete syntax) and abstract representations of the same language instance that uses more abstract symbols that cannot be used in a text but only represent the parts of a language instance necessary to determine its meaning. Anyhow, since the meaning of these terms depends on the used language description method, I will not use these terms in this thesis.

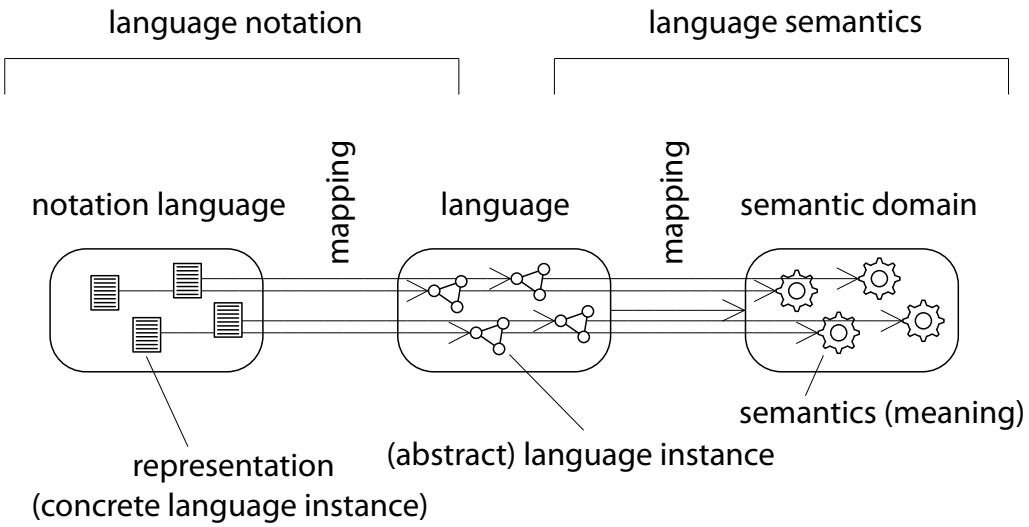


Figure 1.1: Representation and semantics of language instances.

instance has a meaning. This includes that two language instances can have the same meaning, or that the semantic domain contains more elements than needed to give each language instance a meaning.

I already briefly introduced the possibility that, depending on the language description method, a language instance can be something *concrete* or something *abstract*.

Definition 6 (concrete and abstract language instances) *A* concrete language instance *is something that humans can create and comprehend; examples are pieces of text, diagrams, or Excel sheets. An* abstract language instance *is just a structure that not necessarily has a physical form.*

You can look at *meaning* from two sides: a language instance can mean something, or it can be the meaning of something. The second possibility allows to give abstract language instances a concrete representation. When you have a language comprised of abstract language instances (abstract language), you can use a different language that consists of concrete language instances as a notation language for the abstract language. Each concrete instance of the notation language acts as a *representation* of an abstract instance of the abstract language. The abstract language becomes a semantic domain for the notation. We can use the scheme consisting of the concepts language, semantic mapping, and semantic domain not only to explain semantics but also representation.

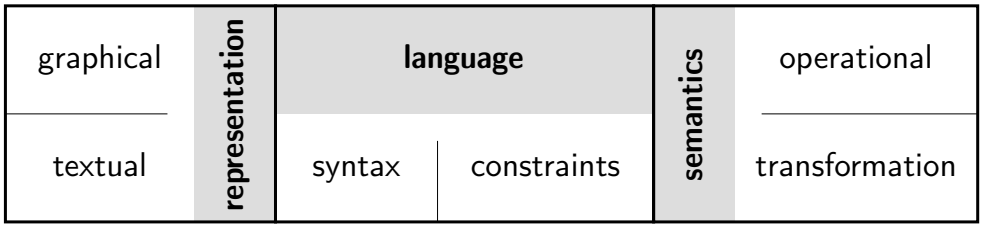


Figure 1.2: A small incomplete hierarchy of language aspects. This diagram displays language aspects and their sub-aspects.

Definition 7 (representation, notation) *A language instance can be represented by another language instance (of another language). This other language instance is a* representation *for the language instance. A language with instances that are representations for the instances of another (the notated) language is called notation language for this other language. A notation language and a mapping from this notation language to the notated language is called* notation*.*

Fig. 1.1.3 illustrates how the concepts language, notation, semantic domain, representation, and semantics work together. Please note, that the scheme consisting of the concepts language, semantic mapping, and semantic domain is not only used to define what semantics is, but also what representation is. Depending on the mapping between notation language and language, you can have multiple representations for the same language instance.

1.2 Language Descriptions

1.2.1 Language Aspects

There is more to a language than just a set of instances: a language is surrounded by language notations that give representation and language semantics that gives meaning. To use a language, not only the language but also representation and semantics are needed. From the structure of the last section, you can already intuitively sense that there are certain *language aspects*.

The Different Language Aspects

The three main aspects introduced so far are *language*, *representation*, and *semantics*.

Definition 8 (language aspect) *The aspect* language *defines what the instances of a language are. The aspect* representation *defines how instances of a language are represented in concrete form. The aspect* semantics *defines what a language instance means (i.e. the information it conveys).*

These main aspects can be further divided into sub-aspects or different kinds of the same aspect. The aspect language, for example, is often divided into sub-aspects *syntax* and *constraints*. Where syntax is described with a constructive description technique that allows to either accept language instances or generate language instances, and constraints are described with a rejecting description technique which allows to reject non-valid language instances. This allows to combine two descriptions: one describes a too large set of language instance containing unwanted instances, the other describes how to narrow the set of instances to those valid, really wanted, instances. The aspect representation on the other hand can be divided into different kinds of representation, e.g. textual and graphical representations. The aspect semantics can be divided into different kinds of semantics. Semantics can define language instance execution (execution or operational semantics), define a mapping into another language (transformation semantics), define a construct by construct translation into the constructs of another language (denotational semantics), and much more [4]. Of course these example sub-aspects and aspect kinds can be divided further . Operational semantics, for example, contains sub-aspects like debugging.

Realisation and Description of Language Aspects

The division into language aspects and sub-aspects is not arbitrary: each language aspect covers a certain use-case of what a language user might do with a language. Reading, writing, or editing language instances is subject to representation; translating or executing of language instances falls into the semantics aspect. The different language aspects, sub-aspects, and different kinds of one aspect correspond to different language tools.

Since each aspect covers its own use-case and corresponds to a different kind of language tool, each aspect also defines its own problem domain. The syntax of a language, its notation, and semantics are all completely different things that require completely different means to describe them. Conclusively, descriptions for different aspects have to be written in different description languages, each description language tailored for the corresponding

[4] This crude attempt to classify differing kinds of semantics or semantic aspects is far from complete.

aspect. As a result each aspect is described separately and for one language you have several descriptions, one for each aspect.

Interconnections between Aspects

Language aspects are described separately and with different means but the corresponding descriptions are not independent from each other. The different description languages for the different aspects form a framework of description languages. The languages in such a framework can be used to describe a language and all its aspects. In such a framework of description languages, the aspect *language* plays a central role. The other aspects, *representation* and *semantics*, define mappings towards and from the set of language instances. Therefore, the aspects *representation* and *semantics* are related to the aspect *language*. More accurately, *representation* and *semantics* are related to *syntax*, which defines the constructs of a language. Notation and semantic mappings refer to the constructs of a language.

Conclusively, descriptions for *representation*, and *semantics* are related to the description of *language*. The used description languages also depend on each other. The description languages can be widely different and independed, but at some point, description languages for *representation* and *semantics* must allow to refer to the construct definitions in a *language* description. Since *language* and more specifically *syntax* is the central aspect, some constructs of the *syntax* description language, have to be addressed by constructs of a *representation* or *semantics* description language.

1.2.2 Existing Language Description Techniques

The idea to describe languages and fully or partially generate language tools from those description goes back to the first programming languages (or higher level languages as they were sometimes classified). Programming languages were in the beginning purely textually notated languages and neither actual nor conceptional separation between representation and language was necessary. At this point, formal language theory derived from linguistics (most notably by Noam Chomsky [14]) was used in an attempt to govern language descriptions. With BNF, Backus created a compact formal syntactic notation based on one of Chompsky's grammars, namely context-free grammars, to describe the syntax of a textual language. BNF provided a link between language theory and practical concerns like syntax analysis of programming language instances. With automatic parser development based on BNF, one could easily create syntax analysis tools that provided the construct structure of language instances represented as term structures typically

represented as *abstract syntax trees* (AST). BNF and context-free grammars allowed to describe a programming language's notation (the set of all grammar recognisable strings, i.e. language instances) and to describe its syntax (i.e. its constructs). Furthermore, BNF allowed to parse language instances into ASTs. Unfortunately, it was only possible to describe the context-free parts of a language and its textual representation; additional constraints to govern non context-free syntax and semantics was unaddressed. These aspects had to be described in other ways, e.g. natural language or general mathematics. Either way, this prevented automatic development of tools covering these concerns.

Several technologies were created to overcome this limitations. I want to mention two of them. Attribute grammars, which were originally introduced by Knuth [56], allow to describe the *semantics of context-free grammars*. Attributes assigned to AST nodes (or language constructs respectively) and evaluation of values for those attributes allowed to give nodes meaning in two different ways. Firstly, attribute values allowed to draw links, transforming the tree into a graph. This allowed to describe reference resolution and type systems so important for programming languages and define constraints to further narrow the set of valid language instances. Secondly, it allowed to compute the overall meaning of a language instance by computing its root AST node attribute values from all its children and children's children. This allows to assign meaning to a language instance and therefore define the semantics of a language.

Graph grammars utilize graph rewriting to describe languages. When the idea of graphical computer languages became popular, a technique to describe and recognise/synthesize graphical language instance was needed. Similar to context-free grammars, which basically describe rewriting on strings, graph grammars describe rewriting on graph structures. Graph grammars also allow to assign semantics to language instance by rewriting them to their meaning and graph grammars are also used to describe context-sensitive parts of textual languages through rewriting of AST's.

Nowadays, BNF, attribute grammars, graph grammars, and other techniques allow to formally describe the presented language aspects and to derive language tool support from descriptions automatically. Theoretically, you can describe and fully automatically develop editors, compilers, analysers, and interpreters, even complete IDEs. Yet, except for context-free grammars and automated parser construction, most of these techniques have not become mainstream. Most of today's compilers, graphical modelling tools, CASE-tools, and IDEs are still manually implemented. Many of today's computer languages are not even formally specified except for syntax and textual notation descriptions based on context-free grammars. I can only

speculate about reasons: attribute grammars and graph grammars are rigorous mathematical formalism that are not easy to master and they are closed formalisms that are not easy to extend once their limitations are reached.

1.2.3 Object-Oriented Meta-Modelling

Object-oriented modelling combines the use of several established system modelling notations (like state charts and entity relationship diagrams) into a coherent modelling method, most widely known through the *Unified Modeling Language* (UML), which itself is most widely known (and used) for its class diagrams. The whole UML methodology and the *model-driven* movement around it defines a new technological space that describes languages not with grammars (originated in linguistics) but with object-oriented modelling itself (originated in systems modelling). In this thesis, I will call the description of languages with object-oriented modelling means as *object-oriented meta-modelling* (OOMM). This is the description of languages with class diagrams; the description of language constructs with classes and relations between classes. While the name object-oriented meta-modelling mainly addresses the language description with class diagrams, I want to understand the term in a broader sense that also includes the description of representations and semantics for languages described with class diagrams.

OOMM allows for object-oriented analysis and design techniques [19, 18] being applied on the subject of language description. Principles, such as reuse, abstraction, inheritance, association, and scale become applicable to the description of existing languages and the design of new ones. Furthermore, object-oriented analysis, design, and programming are a widely spread methods among software developers. Opposite to grammar-based techniques, there are no new underlying principles that have to be internalised; existing developer skills can be directly used for describing languages and creating language tools. Opposite to grammars, OOMM is not tailored for a specific kind of language notation. With OOMM you simply define a set of abstract structures (language instances). OOMM is not intrinsically attached to representation recognition (i.e. parsing) or representation synthesis (i.e. program synthesiser or syntax directed editing). Notations have to be described fully separately. This means on one hand that an additional description technique has to be mastered, on the other hand this means a strong separation of representation and language instance. This allows, for example, to represent a single language instance with different techniques, e.g. combine graphical and textual notation. It also allows unorthodox language representations such as Excel spread sheets or non human, computer gener-

ated representations [5]. Finally, OOMM promotes language descriptions and language instances that are well aligned with object-oriented programming. As the dominating programming paradigm, object-orientated programming is used to create language tools. Even if the language tools are generic or generated, OOMM alignment with object-oriented programming provides an advantage because the development of generic tool or tool-generator requires programming at some point.

It follows a summary of the presumed strengths of OOMM:

- OOMM is easy to master by languages developers. Those specialised engineers are already familiar with object-oriented techniques.
- OOMM allows to manage language complexity based on proven object-oriented techniques, such as abstraction, reuse, or modularisation.
- OOMM governs graphs, which is important to cover all sorts of language instances.
- OOMM's strong separation of language instance and its representation allows all kinds of notations.
- OOMM is well aligned with the predominant object-oriented programming paradigm.

1.2.4 Language Description Requirements for Modern Computer Languages

Modern software engineering does not solely rely on textual programming languages anymore. Methodologies like *Model-Driven Development*, *Product Line Development*, or *Domain Specific Languages* rely on languages with far more diverse characteristics. Furthermore, you use a large number of languages within a single software development project: you use different languages to describe the problem domain or software product in different project phases and on different abstraction levels. As a result, modern computer languages have the following characteristics:

- They are not all textual languages. You have languages that are notated in text, tables, or diagrams. Thereby, a single language instance is often represented by a number of texts, tables, and diagrams.

[5] OOMM is often used to define languages that are used in an intermediate step of a model-driven development transformation chain. In such a case the used language is not used by any human, but only by the computer system executing automated steps in such a development process.

- Languages might use several notations. This can mean that a whole language instance can be represented in different notations, or that different parts of a language instance are represented in different notations.

- Several instances of different languages describe the same thing on different abstraction levels or from different views. As a result, the constructs of different languages are logically related to each other.

- More general languages are used in more specific contexts and have to be specialised for this purpose. This requires means to specialise language constructs or create language profiles to alter language representation and semantics.

- There are very specialised languages with a narrow set of constructs that are only used within one domain (Domain Specific Languages) or even for just a single project.

Not only the character of computer languages has changed, but also the habits of using them and how these languages are realised in language tools. Where in the past a plain text editor and compiler was satisfactory, you now demand integrated development environments (IDE) that combine the tools of all used languages. You switch between different views on your software, between different abstraction levels. Changes made to one part of a software system description automatically changes others. Characteristics of modern computer language tools are:

- The efficiency and quality of software development depends on the quality of language tools as much as it depends on the quality of the used languages.

- Highly specialised languages, such as DSLs, require efficient development of language tools to be economical.

- Specialised languages also require to change a language often. When the domain changes, the DSLs have to be changed as well.

- Tools should be integrated; language instances are exchanged between tools of different vendors; changes made in one tool inflict automatic changes in others.

From those characteristics of computer languages and their use, I can derive a set of requirements for language descriptions.

- Language instance and representation have to be two separate things. Language descriptions must allow the definition of several notations for a language. Different forms of notations have to be combined.

- Language description techniques, including techniques for the description of language, notations, and semantics, must provide the means to express abstractions within language constructs. Language construct definitions should form specialisation hierarchies, including the reuse/inheritance of construct characteristics and related descriptions for representation and semantics.

- Language descriptions must allow the efficient development of language tools and prototypes. It should be possible create language tools based on language descriptions as automated as possible.

- Language descriptions should be well aligned with predominant programming paradigms to support tool development. Although, the aim is fully automated tool development, today's tools are partially created automatically based on language descriptions and partially programmed. Therefore, language descriptions techniques should be well aligned with predominant programming paradigms, especially object-orientation.

- It must be possible to combine different language descriptions. The description techniques must allow to relate language constructs of different languages. Language descriptions must facilitate the description of mappings between different languages.

1.3 Domain Specific Modelling of Language Tools

Computer languages are used to convey information to a computer. This inevitably means that a computer has to be able to process language instances. The existence and quality of the computer tools that can process the instances of a language heavily influence the usability of a computer language.

Definition 9 (language tool) *I call all tools that create, maintain, or process language instances* language tools*, and the set of all tools for a language the* tooling *of this language.*

This section is about the relationships between language descriptions and language tools. In this thesis, I want to use a *Domain Specific Modelling* (DSM) approach to develop language tools based on language descriptions. The use of DSM will be motivated later; for now, DSM promises the automated development of language tools from language descriptions. Language tools thereby can become a effortless by-product of describing a language. Firstly, I introduce DSM in general and secondly apply it to the development of language tools.

1.3.1 Domain Specific Modelling

Instead of manually programming similar programs again and again, *Domain Specific Modelling* (DSM)[6] uses specialised description languages to automatically develop the programs of a class of similar computer programs. These specialised description languages allow to raise the level of abstraction and thereby reduce development efforts. You only describe the things that are specific for a concrete computer program and neglect the things that are common for the whole class of similar programs. In this context, the specialised description languages are referred to as *domain specific languages*, where the term *domain* characterises the specialisation for a specific domain, manifested in a specific class of computer programs. The description themselves are called *models* in this context, emphasising the fact that these descriptions *model* a computer program on a higher-level of abstraction. To apply DSM for a specific description language, you need tooling for this description language that allows to automatically create the described artefacts. I refer to this tooling simply as *tooling* or *DSM tooling*/*DSM tools*.

Description languages are intended to describe a specific class of things. A description language for textual notations, for example, is used to describe the concrete syntax of textual computer languages, or a description language for operational semantics is used to describe the execution of instances of a programming language. Descriptions in general can be used to automatically develop computer programs. This requires tooling for the specific description language. This tooling can create a specific class of computer programs based on the specific class of things that the description language can describe. Diagram editors for graphical computer languages, for example, are very similar. They provide a drawing canvas and a tool box. Users can select tools and create graphical items. They can select objects and move or delete them. All these features are included in all graphical editors. The only

[6]The term was coined by Juha-Pekka Tolvanen[51], more specific approaches limited to generating computer programs from descriptions are Model-Driven Software Engineering[63], or generative engineering[20].

thing that is language specific is the used notation: the used symbols and possible connections. With DSM a language developer only describes the notation and generates the editor with all its common characteristics from that description.

A description can be unambiguous or ambiguous: it can either describe exactly one thing or it describes, deliberately or not, two similar but different things. A description language can have unambiguous or ambiguous semantics: all instances of a unambiguous description language are unambiguous; some instances of an ambiguous description language are ambiguous. Only unambiguous description languages are reasonable for DSM, because only those guarantee that the automatically created computer program reflects the intended meaning.

A description language can use constructs that are either similar or completely different to the computational concepts of a target platform. This influences whether or not DSM for this description language and this target platform is feasible or not. For example, a description language that relies on abstract mathematical constructs that are hard to realise on a computer platform makes it hard to create a DSM tooling for that language. This usually corresponds to the level of abstraction that a description language uses. The more abstract a description is, the more details the corresponding tooling has to create by itself. Since a DSM tooling is written for a description language and not specific for each description, the abstract to concrete mapping solutions that the tooling provides are generic for all descriptions. This either makes it impossible to develop tooling, because creating such details requires more intelligence than one can put into such tools, or the tooling creates computer programs that do not perform well enough. As a general rule of thumb, the more abstract a description language is relative to the target platform, the more intelligence has to be put into the tooling and the more generic and therefore less performing are the automatically created computer programs.

As a conclusion, description languages for DSM can not be arbitrarily abstract. This presents a trade-off: on one hand a description should be as abstract and as small as possible, on the other hand a description has to allow efficient (automatic at best) development of performing computer programs. Therefore, the description languages in today's application of DSM present a compromise. They provide constructs on a fairly high abstraction level. This allows to realise a small amount of the most frequently appearing use cases, but does not cover the very special and therefore seldom details. The argument behind this strategy is that one can describe the bigger part of what one needs to describe and for all the specialities one has to leave the realm of the description language and use a different technique. Most DSM

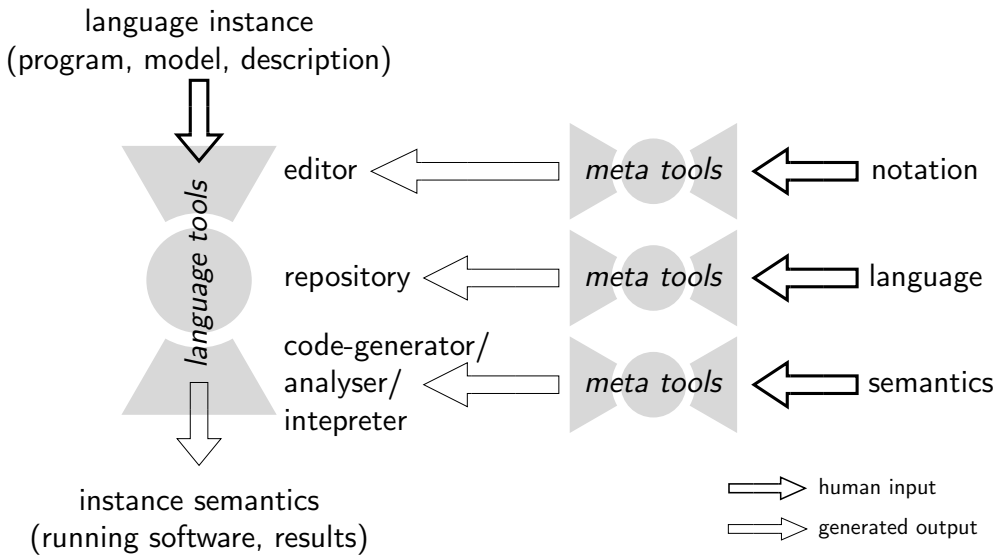


Figure 1.3: Schematic overview of the domain specific modelling of computer languages. Language tools are generated from language description via the use of meta-languages and corresponding meta-tools.

tools therefore allow to augment the instances of specific description languages with pieces of code written in multi-purpose programming languages. This renders the development of corresponding computer programs at least partially automatic. As a result, DSM is very popular for the development of prototypes that do not necessarily need to contain all the specific details that are required for the final software product.

1.3.2 DSM of Language Tools

Along the different language aspects, a language tooling comprises editors that allow language users to create language instances, analysers, code-generators, or interpreters that allow to process and execute language instances. DSM can allow to automatically create language tools from language descriptions.

Language tools fall into several categories in the same way languages comprise several aspects. For each different language aspect you use a different description. With the corresponding DSM tools for each of these description languages, you can create the different language tools for the described language. Fig. 1.3 gives an overview about language tools, language descriptions, and DSM tooling to create the language tools.

A description, no matter if it is a description for a language, represen-

tation, or semantics, is just a piece of information. A way to represent this information is in a language specialised for this kind of information. This becomes confusing, if I neglect to strictly distinguish between different levels of description and described object:

Definition 10 (meta-language) *A* meta-language *is a computer language used to describe other computer languages.*

It is not necessary to write a description in a computer language, but describing a language in a computer language allows you to apply DSM and create language tools automatically. You need to make the same distinction between language tools, and programs that create language tools (DSM tools, meta-language tools):

Definition 11 (meta-tool) *A* meta-tool *is a computer software that generates language tools based on a description written in a corresponding meta-language.*

The relationship between meta-language and meta-tool is the same as between language and language tool. While language tools that realise semantics might simulate, analyse, or compile language instances, meta-tools generate language tools from meta-language instances. The only difference between a common computer language and meta-language (the same holds for tools and meta-tools) is its purpose. A meta-language is just a computer language with the specific purpose to describe other computer languages. Meta-language and corresponding meta-tools form applications of DSM.

1.4 Thesis Motivation, Aim, Hypothesis, and Contribution

There are four reasons that motivate the work in this thesis:

First reason: the nature of computer languages changed dramatically in the last decades. There are two particular popular trends in computer languages. Firstly, graphical modelling languages are not plain pen and paper languages anymore. Graphical modelling has become a major use case for computer languages. Instances of graphical languages are edited, analysed, simulated, or translated. Secondly, opposite to compiling programming languages, MDA and similar modern software engineering approaches require language instances to be simulated, translated into other non-programming languages, or need to generate code from language instances. Therefore, you

need to describe different kinds of language semantics. Such new language characteristics and diversity thereof often requires language engineers to combine existing techniques in unusual ways or to abuse existing technology for new ends (I presented a list of requirements for describing modern computer languages in 1.2.4).

Second reason: there are more and more complex languages and an ever increasing number of languages. In all times in the history of computer science there were reasons for a vast amount of languages, and a need to describe these languages rigorously, not only for the purpose of human understanding, but also the wish to automatically create language tools. Fig. 1.4 illustrates that modern computer languages operate on a higher level of abstraction. Therefore, they are either utterly complex (featuring a large number of language constructs) or they are very specific. Being specific allows them to be small but requires a large number of new languages. In both cases, you have to develop languages and language tools for a vast number of language constructs. DSM could be used to handle this amount of work economically. DSM of language tools promises to create DSL tooling efficiently enough for an economic application of DSLs, and the object-orientated nature of OOMM might allow to govern the complexity of languages (more details about DSM and its potential are discussed in 1.3).

Third reason: manual implementations of languages often cause a gab between language description and language implementation. This is most often evident for bigger languages, e.g. UML or SDL, which are extensively specified before tools are implemented by several vendors. As a result of manual implementation of language tools, the tools do not always precisely reflect the language as described. Instead, language tools implement the specification (description) incompletely, incorrectly, or not precisely. Language users cannot rely on the language specification (description), but have to refer to what the used tooling does instead. Examples for this phenomenon are the significant differences between existing UML tools, or the non-existence of actual SDL-2000 tools (even though the SDL-2000 standard is almost 8 years old). With DSM of language tools and its ability to at least partially generate language tools from language specifications, it is thinkable to significantly lessen this problem.

Forth reason: object-oriented meta-modelling presents a new technological space. I already introduced object-oriented meta-modelling and its possible advantages in 1.2.3. Object-oriented meta-modelling (OOMM) is intrinsically different from existing language description technology based on grammars. While grammar-based language description techniques were

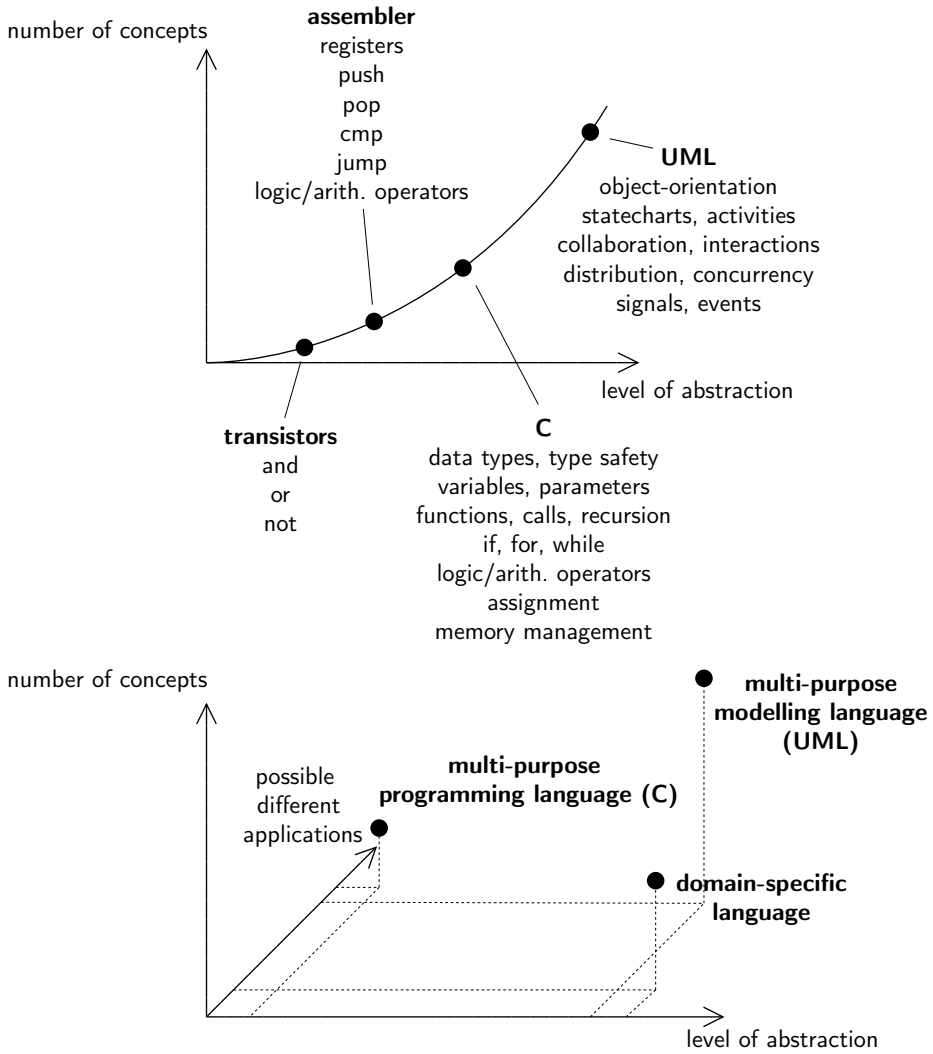


Figure 1.4: Over time languages were used on a more and more abstract level of abstraction. As a result, languages grew more and more complex. One solution to this problem is the use of specialised domain specific languages. As a result, you are confronted with a need for bigger and/or frequently new languages, which have to be developed more and more efficiently.

thoroughly researched over nearly 50 years, OOMM is only a decade old. Although the idea to use OOMM for describing a computer language, including aspects like representation and semantics, certainly exist, it is not yet researched. It seems possible and, judged by OOMM's supposed strength, plausible that describing computer languages with OOMM is indeed a reasonable idea. Furthermore, MDA and MDSD are founded based on OOMM and need language descriptions based on OOMM to work.

Lets summarise the four motivational points. Firstly, today's languages have very diverse characteristics, which requires a cohesive frameworks of language description languages based on a single methodology like OOMM. Secondly, there is a growing number of DSLs and languages with increasing complexity. DSM could be a solution to decrease the resulting manual labour, and object-oriented meta-modelling could be a solution to govern complexity. Thirdly, there can be gabs between language descriptions and tool implementations, which again could be closed by DSM of language tools. Finally, OOMM is a fairly new technology and its potential is not yet fully researched.

These are all good reasons to research whether OOMM can be used to describe languages in all important aspects, and whether DSM can be used to create language tools automatically from OOMM based language descriptions. Thereby, the reason to show that OOMM can be used to describe languages fully is not to show that OOMM might provide better means to describe languages than grammars. Nevertheless, grammar-based techniques play an important role even for the description of language with OOMM. Describing textual notations for example obviously suggest the use of grammars. Furthermore, you can use graph transition systems (e.g. graph grammars) to manipulate language instances described with OOMM, for example to describe semantics. For all these reasons:

> *It is the aim of this thesis to apply OOMM-based techniques to all important aspects of language descriptions and thereby reason about how well suited OOMM-based techniques are to describe modern computer languages.*

Because computer languages can take a huge variety of different shapes, I have to narrow them down to a manageable subset of languages considered in this thesis. I take a specific language as archetype. I choose SDL as this concrete language example. SDL is a complex, textually and graphically notated language with a clearly defined operational semantics. Other language examples that fall under this archetype are all imperative programming languages or modelling languages based on Petri-nets or state-charts. In parts, the used techniques are also applicable to other languages: for example, ar-

bitrary notated languages with operational semantics (you can still use the techniques to describe the semantics) or textual notated languages with a different kind of semantics (you can still use the techniques to describe textual notations). To see in the end, if I could reach the aim, I try to prove the following hypothesis:

> *The hypothesis of this thesis is that all aspects of the SDL language can be described with specialised meta-languages and the resulting descriptions of SDL allow to generate a prototypical toolchain for SDL.*

To prove this hypothesis, I examine existing OOMM-based meta-languages and corresponding meta-tools. If necessary, I improve them or create new ones. I want to show how far OOMM can be used for a complete description and automatic language tool generation. The contribution of this thesis therefore is not only to show that SDL can be described thoroughly and that DSM can be applied to these descriptions, but also the design and implementation of new meta-languages and meta-tools. For each language aspect, I present existing and ongoing research and present own approaches, meta-languages, and meta-tools, which contribute to the OOMM community and enhance OOMM-based language descriptions and DSM of language tools in general. This includes the following three new original scientific contributions[7]:

First contribution: the means for a better utilisation of object-orientation in OOMM than existing technology allows. I developed a OOMM-platform that, for the first time, implements the new object-orientation features of the latest OMG MOF 2.x recommendations. As you will learn later, these features allow to significantly increase the expressiveness of language descriptions when it comes to language construct specialisation and therefore help to deal with the complexity of language descriptions.

Second contribution: New meta-languages and meta-tools for textual representation and operational semantics with unique characteristics. The meta-language and meta-tool that I developed to describe concrete textual representations allow to automatically create feature-rich textual editors that support, for the first time, content assist for named references and embedding textual editors into graphical editors. The meta-languages and meta-tool for operational semantics, in contrast to existing

[7] A more detailed list of contributions, including references to publications, can be found in the conclusions (6.1).

technology, combine existing system modelling languages that engineers are already familiar with.

Third contribution: A case-study for OOMM-based language descriptions and DSM based a real language. Most case-studies for OOMM-based DSM of language tools are limited to a certain language aspect or are made for small DSLs. I describe and create the tools for the important parts and all important aspects of a complex industry scale system design language (SDL). I thereby show the principle applicability of OOMM-based DSM for languages of this size and complexity.

1.5 Thesis Outline

Including this introduction, this thesis has six chapters. Chapters two to four discuss OOMM-based language descriptions, aspect by aspect: language, representation, and semantics. Chapter five applies the presented techniques to the SDL language, and the sixth chapter draws conclusions on how well suited meta-modelling techniques are to describe SDL in particular and modern computer languages in general.

The three chapters about the different language aspects each introduce the necessary concepts, meta-languages, and meta-tools to realise the discussed language aspect. Each of these chapters features an introduction, related work, and conclusion in its own right. The chapter on language and syntax provides a formal model for MOF-like meta-modelling, and a programming framework that allows to maintain meta-model-based language instances in computer memory. This framework provides the basic means to program with language instances and to develop language tools. The following chapter on textual representations gives a formal comparison of grammar-based and meta-model-based language descriptions with the goal to use grammars as a technique to describe textual notation for meta-model-based languages. The chapter features a notation description language based on the context-free grammar formalism and presents a meta-tool that allows the DSM of textual model editors. The chapter on semantics shows a way to describe operational semantics in a human comprehensible yet machine processable way. The presented meta-language for operational semantics and corresponding meta-tool allows to describe the semantics of a language and allows to execute language instances right away.

The chapter on the SDL case-study presents experiences in applying the presented meta-model-based techniques to SDL. It shows the capabilities, advantages, and problems with meta-model-based language descriptions.

Conventions

In the mathematical parts of this thesis, I use the following conventions.

For a relation $rel \subseteq A \times B$, I often write $rel(a)$ to denote all elements related to a $a \in A$:

$$rel(a) = \{b | (a, b) \in rel\}$$

For a more complex relation $Tuples \subseteq A_1 \times A_2 \times \cdots \times A_n$ each element $tuple \in Tuples$ has the form

$$tuple = (label^1, label^2, \ldots, label^n)$$

and I write $label^i_{tuple}$ to denote the i-th position of $tuple$. To define a left-total and right-unique relation $func \subseteq A \times B$, I use the common notation for functions: $func : A \to B$. I use $func(a)$ for each $a \in A$ to denote the one unique $b \in B$ that a is related to.

The set $\mathbb{S}$ is the set of all possible strings that make suitable names for classes and properties, $\mathbb{B}$ is a set of boolean values $\mathbb{B} =_{def} \{true, false\}$, $\mathbb{N}$ is the set of natural numbers, $\mathbb{N}^\star$ is the set of natural numbers including a element for infinity: $\mathbb{N}^\star =_{def} \mathbb{N} \cup \{\star\}$ with $a \leq \star$ for all $a \in \mathbb{N}^\star$.

Definitions in grey boxes that have the annex *math.* in their titles (they look like **Definition 5 (Title, *math.*)**) give a mathematical definition for terms that are already introduced, defined, or used otherwise.

Chapter 2

Languages – Describing Languages with Meta-Models

2.1 Introduction

In this chapter, I provide the foundations for the description of languages. The means necessary to describe what the instances of a language are and how these instances can be created and maintained within a computer are provided. These are the foundations for describing computer languages and for developing language tools.

I start this chapter introducing the notion of a *meta-modelling platform*: the conceptual and technological basis for language descriptions. I will introduce MOF-like modelling platforms—object-oriented meta-modelling platforms that I will use as a specific language description technology used throughout this monograph. After a brief presentation of concrete existing MOF-like platforms, I discuss problems in these platforms. This is followed by the presentation of related work suggesting solutions to these problems. The remainder of this chapter is dedicated to problem solutions and my modelling platform *A MOF 2 for Java* that realises this solutions.

2.2 Meta-Modelling Platform

Definition 12 (meta-modelling platform) *A* meta-modelling platform *provides all languages, frameworks, and tools necessary to describe language constructs and develop language tools. A meta-modelling platform uses a* meta-modelling formalism *and consists of a* meta-modelling framework *based on this formalism.*

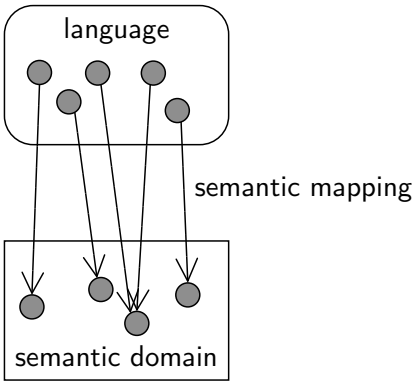


Figure 2.1: An example formalism, consisting of language, semantics, and semantic domain.

To realise such meta-modelling platforms, I have to introduce some underlying concepts first. Firstly, there are meta-modelling formalisms that I use to define languages and their constructs (section 2.2.1). Based on that, I introduce meta-modelling frameworks, which are programming frameworks that I use to build language tools based on languages described in a meta-modelling formalisms (section 2.2.2). Finally, I put everything together and discuss how meta-modelling platforms combine meta-modelling formalisms and frameworks into a conceptual and technological foundation for the description of languages and development of language tools (section 2.2.3).

2.2.1 Meta-Modelling Formalisms

Languages and Formalisms

A *language* is a possibly unlimited set of language instances that fit a language description. A language can be associated with a *semantic mapping* and *semantic domain*. A *semantic domain* is a possibly unlimited set of elements. Each of this elements is a possible *meaning* of a language instance. A semantic mapping is a total mapping between a language and a *semantic domain* that assigns a *meaning* from the semantic domain to each language instance.

Definition 13 (formalism) *A structure $F = (L, SM, SD)$ consisting of a language L, a semantic domain SD, and a semantic mapping between the language and the semantic domain SM is called a* formalism.

Fig. 2.1 shows the three elements of a formalism. Examples for concrete formalisms can be taken from Petri-nets [88]. The set of all place-transition

Petri-nets without markings is a language. A possible semantic domain could be the set of mappings between markings and sets of sequences of firing transitions. Another semantic domain is the set of mappings between markings and *reachability graphs*. Another example is the Java programming language in conjunction with a Java compiler. The language here is the set of all statically correct Java programs. The semantic domain is the Java byte-code language (a set of all statically correct byte-code sequences), and the semantic mapping is defined by the inner workings of the Java compiler [38, 62].

A formalism can be used to describe the semantics of a language. But, a notation is just a language with another language as semantic domain. Hence, formalisms can also be used to describe the representation of a language. The notion of a formalism and its constituents is an abstract conceptualisation: it abstracts from concrete language instances and concrete semantic domains. At this point, you are neither limited to a certain kind or complexity of languages nor a certain kind of semantic domain. It is also not fixed how any of the formalism components can be described.

Meta-Modelling Formalism

A language, as a set, can be defined in two ways. You can either define a language *extensional* by enumerating the language instances, or *intensional* by specifying the necessary and sufficient conditions for language instances to be instances of the language. While the first opportunity can be used to reason about language definitions it is certainly not practical, since most languages have an unlimited amount of instances. The introduced definition of the term *language description* defines a language description as an intentional language definition: a language description defines a language intensional.

I defined that a language description consists of language constructs definitions, which describe the syntax of a language. But what are possible language descriptions, language construct definitions and how do these entities look like? I use specific formalisms that I call *meta-modelling formalism* to define what syntaxes are and what they mean. Fig. 2.2 shows the constituents of a meta-modelling formalism.

Definition 14 (meta-modelling formalism) *A* meta-modelling formalism *is a formalism that has a set of languages as semantic domain. The language of a meta-modelling formalism defines a set of possible language descriptions. The language of a meta-modelling formalism is called* syntax description language. *The semantic mapping of a meta-modelling formalism defines which language is defined by a syntax of this formalism. The domain of the semantic mapping is called the* set of generated languages *of the meta-modelling formalism.*

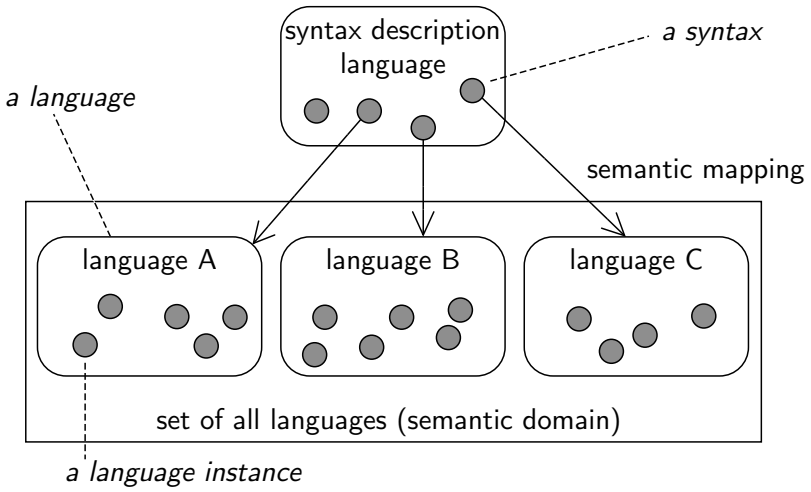


Figure 2.2: A meta-modelling formalism.

A concrete example for meta-modelling formalisms are BNF-grammars [1, 44]. The language of the BNF-grammar formalism is the set of all BNF-grammars over a given set of terminal and non-terminal symbols. The semantic domain is the set of all context-free languages that can be formed over the given set of terminal symbols.

The notion of a meta-modelling is an abstract conceptualisation: it abstracts from concrete formalisms. At this point you are not limited to concrete types of syntaxes or concrete types or complexities of generated languages. However, once you choose a concrete syntax formalism, it only allows you to define the generated languages. As you see from the examples, these are usually limited classes of languages, like context-free languages in the BNF syntax formalism.

Describing a Meta-Modelling Formalism and Self-Contained Meta-Modelling Formalisms

You now know what a meta-modelling formalism is, but how can you describe a concrete meta-modelling formalism? I will use simple set-theory and logical formulas to describe the language, semantic mapping, and semantic domain of concrete meta-modelling formalisms. This will be done for MOF-like meta-modelling and context-free grammars. However, there is another way to describe the syntax description language of a meta-modelling formalism.

You can use another meta-modelling formalism to define the syntax description language. Or, you simply use the meta-modelling formalism itself to define its own syntax description language. The meta-modelling formalism's

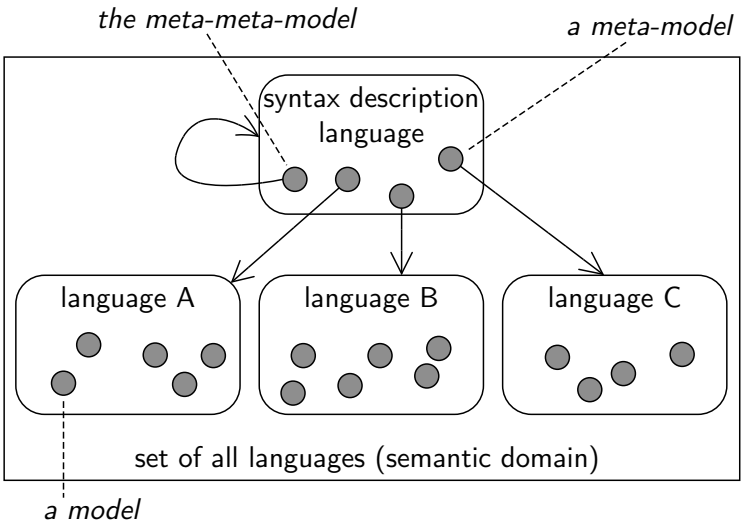


Figure 2.3: A self-contained meta-modelling formalism with modelling terminology.

language can be part of its own semantic domain. Furthermore, there can be one or many syntaxes in the syntax description language that are mapped to the syntax description language itself. In the BNF-grammar formalism example, there is at least one BNF-grammar that defines a language that contains all BNF-grammars. I say a meta-modelling formalism is *self-contained*, if the syntax description language is an element of the semantic domain and there is at least on syntax that is mapped to the syntax description language.

Formalisms are often wrongly labled self-contained. But in these cases, the meta-modelling formalism is at least generating a superset of the used syntax description language. MOF-like formalisms, for examples, are called self-contained, even though the proclaimed syntax of the syntax description language generates more than the actual syntax description language. The problem is that the definition of the MOF syntax description language relies on constraints. Those constraints have more expressiveness than the MOF syntax description language itself. As a result, the set of generated languages contains also instances that violate these constraints. I call this form of self-containment *weakly self contained*, and I call self-contained formalisms also *strongly self-contained*. In order to create a strongly self-contained MOF formalism, you have to combine a MOF formalism with a formalism for expressing constraints, e.g. the *Object Constraint Language* (OCL). Here you would leave the realm of language descriptions that only contain language syntax: you need to combine syntax and constraints to fully describe the syntax description language of a MOF formalism.

Terminology in the Context of Modelling – Meta-Modelling Formalisms

In this thesis, I will mainly deal with meta-modelling formalisms that are based on concepts recommended by the OMG within the different versions of the *Meta Object Facility* (MOF). Since the MOF technology, originally designed to maintain meta-data in CORBA, was soon tailored to define the *Unified Modelling Language* (UML), most of the used terminology is derived from modelling, especially *model-driven software development* (MDSD)[1]. MOF provides specific means to describe languages and uses specialised concepts for some of the general terms introduced so far. I will use the MOF-specific terms to avoid confusion when I talk about different technologies for language descriptions.

Definition 15 (*modelling terms*) *In the context of MOF meta-modelling formalisms, the instances of a language are* models*, the languages are* modelling languages*, and the syntax description language is a* meta-modelling language*.*

A syntax, based on the semantics of the defining formalism, depicts a set of models. Therefore, a syntax is a meta-model*. The one syntax that describes the syntax description language is a* meta-meta-model*, because it depicts the meta-modelling language itself.*

A meta-model defines the constructs of a language. The construct instances in a model are model elements*.*

Models, meta-models, and meta-meta-models form *model layers*, where each model on one layer is an element of the language described by the model on the layer above: the top layer is the layer containing the meta-meta-model (M3-layer), the layer below contains meta-models (M2-layer), the lowest layer contains models (M1-layer).[2]Fig. 2.3 shows an example for such a meta-modelling formalism.

When I talk about formalisms, languages, and language instances, I am not necessarily interested in the specific application of these concepts. But, the modelling terminology is application-driven: language instances are

[1]The most notable representative of a concrete model-driven software development methodology is the *Model-Driven Architecture* (MDA), which is directly based on OMG recommendations like MOF and UML. The term MDA must not be confused with MDSD. A common synonym for MDSD is *Model-Driven Development* (MDD), which also must not be confused with MDA.

[2]This layered structure is often called 4-layer architecture: the 3-layers are extended on the lower end with an additional M0-layer. The M0-layer can either be interpreted as all the objects that are described or *modelled* in the M1-layer, or more specifically as all the instances of the modelled (software) system.

M3	BNF Language	XML-Schema Language	MOF-Model
M2	BNF Grammars	XML-Schemata	Meta-Models
M1	Sentences of an alphabet	XML-Documents	Models
instances	parse-trees, derivations	DOM (document object model)	directed attributed graphs

Figure 2.4: Three different meta-modelling formalisms.

called models, because languages are used to *model* software in MDSD. But as it turned out, MOF and similar technology can and is also used for languages with other purposes. This leads to a misleading vocabulary within the community. The term model, originally meaning an abstract description of something (in MDSD usually a software system or part of a software system), is often used to refer to all kinds of language instances.

Examples for Meta-Modelling Formalisms

There are several well known meta-modelling formalisms; Fig. 2.4 shows three of them and names their meta-meta-model (M3), their meta-models (M2), and their models (M1).

2.2.2 Meta-Modelling Frameworks

In the last section, I provided the basic concepts for the description of languages. I defined what a *meta-modelling formalism* is and how it can be used to describe new languages. In this section I discuss how you can facilitate meta-modelling formalisms with *meta-modelling frameworks*.

Definition 16 (meta-modelling framework) *A* meta-modelling framework *(or short* modelling framework*) is a programming framework*[3] *that* realises *a meta-modelling formalism. As such, a modelling framework provides a meta-meta-model, can process meta-models (it provides a syntax description language), and realises the semantics of the meta-modelling formalism. This functionality is provided as program libraries and auxiliary tools. This means a modelling framework provides the tooling for its meta-modelling language.*

Modelling frameworks can be used to develop language tools based on a language description given in the realised meta-modelling formalism.

[3] In [36] a programming framework is defined as: *"A framework is a set of classes that make up a reusable design for a specific class of software."* I use this definition of the term, but exchange *classes* with software libraries for generality.

Basic Functionality: Programming with Language Instances

Language tools read, manipulate, and create language instances. Therefore, a good modelling framework provides everything necessary to allow efficient programming with language instances. There are three components that modelling frameworks have in common: they hold language instances in computer memory, provide a language-dependent and language-independent interface to access and manipulate a language instance in memory.

In order to represent arbitrary language instances in computer memory, the modelling framework has to provide a data structure that realises the semantic domain of the meta-modelling formalism. This data structure can be used to store any language instance and its relation to its defining meta-model, independent of the actual meta-model.

Definition 17 (repository) *The framework component that consists of this data-structure and functionality to modify data in this structure is called a* repository component*. An instantiation of this software component in computer memory is called* repository[4]*.*

A repository can be used to store one or more language instances in computer memory. Each language instance separate. To keep each language instance separate in memory, frameworks use *extents*.

Definition 18 (extent) *Within a modelling framework the boundaries between distinct language instances are formed by* extents[5]*. Each extent consists of one instance of the underlying data structure and therefore holds one language instance. One repository can contain multiple extents.*

Repositories can be accessed with APIs. These interfaces can be used to program tools based on a repository. I distinguish two different kinds of interfaces: *meta-model-independent interface*[6] and *meta-model-dependent interface*[7]. Meta-model-independent interfaces are used for tools that are

[4]The term repository was formed by early MOF frameworks, like [70], which considered persistent model storage as the main application. From there on the term repository was also used to refer to transient model storage.

[5]The term *extent* is used in MOF [77, 79]. In EMF [117] the concept is called *editing domain*. This term refers to the fact that an *editing domain* is associated with additional functionality needed to edit a language instance.

[6]This kind of interface is often referred to as *reflection interface*. When using a reflection interface, a program has to introspect (i.e. reflect on) the meta-model at runtime in order to analyse a model, because the meta-model is unknown at programming time.

[7]This kind of interface is often referred to as *generated interface*, because it has to be generated from a meta-model.

independent from a concrete meta-model. These are often tools that can be parametrized with a meta-model at runtime. I call such tools *generic tools*. Meta-model-dependent interfaces are generated from a specific meta-model and only allow to access instances of this meta-model. Meta-model-dependent interfaces allow more efficient and statically safe development of tools, since they use vocabulary defined in the meta-model and allow static checking based on information in the meta-model. Both kinds of interfaces are based on the concepts used in the underlying meta-modelling formalism. The set of rules that relates the formalism concepts to corresponding interface constructs of a programming language is called *language mapping*.

Definition 19 (model interface) *A* meta-model-independent interface *allows to read and manipulate a language instance represented in a repository without knowledge of the corresponding meta-model. A* meta-model-dependent interface *allows to read and manipulate a language instance represented in a repository based on vocabulary defined in the corresponding meta-model. A* language mapping *determines how the constructs of a meta-modelling language are mapped to a programming language and thereby determines the design of both interface kinds.*

Auxilary Functionality

Besides the main three components, modelling frameworks provide other functionality based on the specifics of the underlying formalism and the targeted applications. One example is the serialisation and de-serialisation of language instances (i.e. repository or extent contents). This allows to store and load language instances to and from files in formalism or framework specific exchange data formats.

2.2.3 Meta-Modelling Platforms

The term meta-modelling platform is just a word to refer to both a meta-modelling formalism and an implementing meta-modelling framework. Thus, a meta-modelling platform is characterised by a specific meta-modelling formalism, it uses on meta-modelling framework, and is based on a concrete programming platform.

There is a multitude of existing and thinkable meta-modelling platforms with certain advantages and disadvantages. Platforms differ in the used meta-modelling formalism. The choice of a concrete formalism determines the paradigm and concepts that can be used to define a language (e.g. the syntax description language) and also influences the specific class of languages

that can be developed (i.e. generated languages of the formalism). The chosen framework usually determines a concrete programming platform. This can be the programming language, or can also involve a specific platform for distributed applications (e.g. CORBA), or specific rich client platform (e.g. eclipse). The framework therefore depicts the used programming paradigm and also influences the choice for a software development methodology.

There are several roles involved in the process of developing and using languages and language tools based on modelling frameworks.

Definition 20 (*roles*) *The* meta-modelling platform developer *creates the meta-modelling platform. He or she defines the meta-modelling formalism and develops the corresponding meta-modelling framework. The* language developer *creates languages. He or she defines languages based on a given meta-modelling platform and develops tools for this language. The* language user *or* software developer *uses language and language tools to develop a software application. The* application user *uses a software application.*

2.3 MOF-like Meta-Modelling Platforms

So far I discussed meta-modelling platforms in general, independent of the nature and specific characteristics of the used meta-modelling formalism and the provided programming frameworks. But, most existing platforms are tailored for a specific domain of application: grammars are used for languages with textual syntax, XML is used for information exchange and interoperability, or data-bases are used for mass information storage and efficient data access.

MOF-like platforms originate in MDSD. The original intent was to store, analyse, and present the meta-data of building blocks of software system, for example, additional information and associations between package, classes, fields, and description of object-oriented or component-based software. With UML's demand to define the syntax of a graphical language, MOF-like platforms found their most prominent application, the description of abstract modelling language syntax.

The characteristics of MOF-like platforms are an object-oriented syntax description language and a semantic domain that is comparable to directed labled graphs. This comprehends two advantages for the description of tool construction for modelling languages. Firstly, it allows flexible language descriptions. Object-orientation allows highly modularised and reusable meta-models, it allows for easier language integrations based on their meta-model, and aligns well with the underlying object-oriented programming platforms. Secondly, its semantic domain allows to cover graphical languages (graphs),

as well as textual languages (trees). The generality of graphs as data structures also suggests that MOF is applicable to other, possible undiscovered, language forms. The success of MOF is evident in the definition of the UML-language, its role for the development of integrated programming environments, and several programming frameworks for the development of domain specific languages. In the remainder of this monograph, I use the terms object-oriented meta-modelling, meta-modelling platform, meta-modelling formalism, or meta-modelling framework to refer to MOF-like platforms, formalisms, and frameworks.

2.3.1 Existing MOF-like Meta-Modelling Platforms

There are a variety of MOF-like platforms. I give a brief introduction into MOF1.x, EMF, and MOF2.x to characterise the state of the art in this field. I also use these to discuss the problems with existing modelling platforms.

MOF 1.x is a modelling platform recommended by the OMG [77]. It is the first commonly accepted modelling platform based on an object-oriented meta-language. Its history provides the bigger part of the motivation for meta-modelling. It was originally created as a platform to describe the modelling language UML in a more profound way. MOF also provides a basic description of a meta-modelling programming framework: it describes programming interfaces in the programming language-independent language IDL [72].

The Java Metadata Interface (JMI) [75] presents a Java language mapping for the MOF 1.x recommendation, and thus specifies meta-modelling frameworks for the Java programming language. JMI simply maps the given IDL interfaces to Java. JMI is implemented, among others, by SUN's Metadata Repository [70] and the ModFact project [82]. The JMI sets many standard rules, which are also used in other MOF-to-Java language mappings.

The Eclipse Modelling Framework (EMF) [117] is a modelling platform tailored for the use based on the *eclipse* platform and *Java*. It tightly connects modelling formalism and model programming framework; it does not feature the necessary abstraction to map EMF as modelling formalism onto another programming platform. EMF has evolved into a quasi standard; it is widely used and accepted. The success of EMF is founded on its Eclipse integration and a multitude of frameworks and tools that have evolved around EMF. This renders EMF, despite or maybe because of its very simple meta-language *Ecore*, a very powerful basis for modelling tools.

With the next major release of UML, there also is a new major release of MOF. The corresponding MOF 2.x recommendations define two meta-modelling languages: CMOF and EMOF. CMOF is used by the UML 2.x

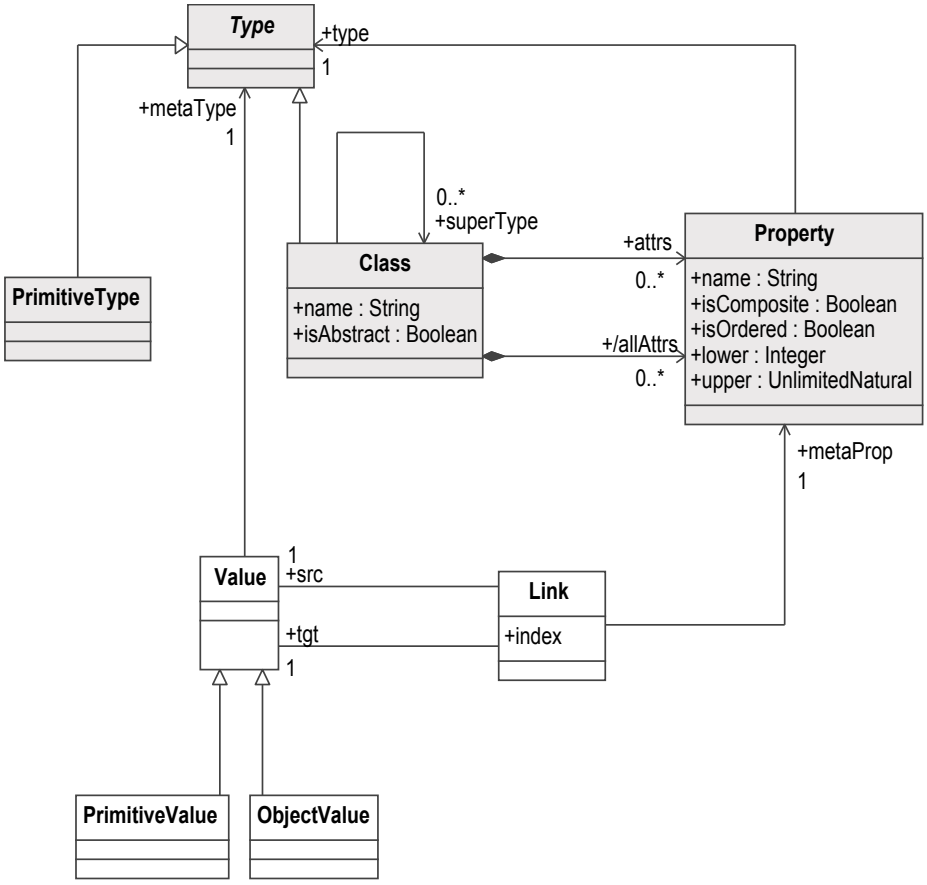


Figure 2.5: A MOF-like meta-meta-model and an instance model.

recommendations to define the new UML. In contrast to older MOF versions, EMOF, and comparable frameworks like EMF, CMOF introduces new features, which raise its expressiveness drastically (refer to section 2.5).

2.3.2 A Formal Definition of a Simplified MOF-Formalism

In this section, I provide a formal definition of a meta-modelling formalism that reflects the essence of a MOF-like formalism. It is not reasonable to try to cover a whole MOF standard formally, but this small definition gives as a valuable tool to analyse and discuss the further problems in this monograph.

Fig. 2.5 shows the core concepts of a simplified MOF-like meta-modelling language (grey, at the top), i.e. MOF-like meta-meta-model. This class-diagram presents the constructs that can be used to define a language in

MOF; each instance of the model represented by this class-diagram is a MOF meta-model. This meta-meta-model is a sub-set of the meta-meta-models in the actual MOF standards and can also be interpreted (by changing names) as a sub-set of the Ecore model. The lower part of the same figure (white) shows concepts that describe an instance description for language instances in MOF-like formalisms. These classes describe the structures used to store instances of a MOF meta-model. In other words this class-diagram describes both, classes that can be used to define language constructs (grey), and classes that are used to instantiate the defined constructs (white).

Meta-models

The class-diagram in Fig. 2.5 provides a conceptual blueprint for the following set-theoretic definition of meta-models.

Definition 21 (meta-model, *math.*) *A meta-model* $\mathfrak{M}$ *is a tuple:*

$$\mathfrak{M} = (\mathit{Types}, \mathit{Classes}, \mathit{Props}, \mathit{superTypes})$$

with

$$\begin{aligned} \mathit{Classes} &\subseteq \mathit{Types} \\ \mathit{Classes} &\subseteq \mathbb{S} \times \mathbb{B} \\ \mathit{Props} &\subseteq \mathbb{S} \times \mathit{Classes} \times \mathit{Types} \times \mathbb{B} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N}^\star \\ \mathit{superTypes} &\subseteq \mathit{Classes} \times \mathit{Classes} \end{aligned}$$

Tuples $\mathit{class} \in \mathit{Classes}$ *and* $\mathit{property} \in \mathit{Props}$ *have the form:*

$$\begin{aligned} \mathit{class} &= (\mathit{name}, \mathit{isAbstract}) \\ \mathit{property} &= (\mathit{name}, \mathit{owner}, \mathit{type}, \mathit{isComposite}, \mathit{isOrdered}, \mathit{lower}, \mathit{upper}) \end{aligned}$$

All the set names and names within the tuples correspond to the classes and properties in the simplified MOF meta-meta-model given in the grey part of Fig. 2.5. It is obvious how a MOF meta-model (an instance of the given simplified MOF meta-meta-model) can be put into a meta-model $\mathfrak{M}$, or vice versa: how $\mathfrak{M}$ denotes an instance of the grey part of Fig. 2.5.

The following left-total helper relations will help to shorten further definitions and formulas:

$$\mathit{allSuperTypes} \subseteq \mathit{Classes} \times \mathit{Classes} \tag{2.1}$$

$$
\begin{aligned}
allSuperTypes(c) &=_{def} \{st | st \in superTypes(c) \vee \\
&\qquad \exists c' \in superTypes(c) : st \in allSuperTypes(c')\}
\end{aligned}
$$

$$
\begin{aligned}
allProps &\subseteq Classes \times Props && (2.2)\\
allProps(c) &=_{def} \{p | owner_p = c \vee \\
&\qquad \exists c' \in allSuperTypes(c) : owner_p = c'\}
\end{aligned}
$$

$$
\begin{aligned}
assignables &\subseteq Classes \times Classes && (2.3)\\
assignables(c) &=_{def} \{c' | c = c' \vee c' \in allSuperTypes(c)\}
\end{aligned}
$$

The relation *allSuperTypes* (2.1) simply extends *superTypes* and also relates indirect super types (the super types of super types and so on) with each other. This is important, for example, to calculate all owned and inherited properties of a class: *allProps* (2.2) gives you all these owned and inherited properties of a class. A property of a certain type can hold values of exactly this type, but also of all sub-types of this type. With *assignables* (2.3) I can denote all types of values that a property of a certain type can hold.

For meta-model $\mathfrak{M} = (Types, Classes, Props)$ to be a valid MOF meta-model certain constraints must be hold. For all $c, c_1, c_2 \in Classes$, $p \in Props$ the following formulas must evaluate to *true*:

$$\forall p_1, p_2 \in allProps(c) : name_{p_1} = name_{p_2} \Rightarrow p_1 = p_2 \quad (2.4)$$

$$name_{c_1} = name_{c_2} \Rightarrow c_1 = c_2 \quad (2.5)$$

$$c \notin allSuperTypes(c) \quad (2.6)$$

$$lower_p \leq upper_p \quad (2.7)$$

The first formula (2.4) ensures that properties of one class must have unique names, this includes all inherited properties. Please note that there is no property redefinition in this simplified MOF. The next formula (2.5) allows only unique class names within a meta-model. Furthermore, super types must not have cycles (2.6), and lower and upper bounds must be reasonable (2.7).

Models

Based on a meta-model $\mathfrak{M}$, I can define what a model is. The white classes in Fig. 2.5 show a class diagram for a model and how the model constructs relate to meta-model constructs. Again, I simply transcribe the names of classes and properties into the following definitions.

Definition 22 (model, *math.*) *A model is a tuple*

$$M = (MetaModel, Values, Links, metaType, metaProp)$$

with:

$$\begin{aligned} MetaModel &\quad \text{is a meta-model} \\ Links &\subseteq Values \times Values \times \mathbb{N} \\ metaType &: Values \to Types_{MetaModel} \\ metaProp &: Links \to Props_{MetaModel} \end{aligned}$$

A tuple link ∈ Links has the form

$$link = (src, trg, index)$$

Each model M can denote an actual instance of the corresponding meta-model. A model M is an instance of meta-model $\mathfrak{M}$ if $MetaModel_M = \mathfrak{M}$. Each value instantiates a type (a class or a primitive type), and each link instantiates a property. Of course the meta-model constrains the ways in which links can connect values. Before I come to the constraints, I define the following left-total helper relations:

$$components \subseteq Values \times Values \tag{2.8}$$
$$components(v) =_{def} \{v' | \exists l \in Links : l = (v, v') \wedge isComposite_{metaProp(l)}\}$$

$$allComponents \subseteq Values \times Values \tag{2.9}$$
$$allComponents =_{def} \{v' | v' \in components(v) \vee \exists v'' \in components(v) : v' \in allComponents(v'')\}$$

With *components* (2.8) you can access all values that are linked to a value based on a composite property. This is important to define the semantics of composition later. You will later see that composition is transitive, therefore I also defined a function for indirect components $allComponents$ (2.9), which also includes the components of components and so on.

For each model M to be a valid model of meta-model $\mathfrak{M} = MetaModel_M$, certain constraints must hold. For all $l, l_1, l_2 \in Links$, $v \in Values$, $c \in Classes_{MetaModel}$, and $p \in Props_{MetaModel}$ the following formulas must evaluate

to *true*:

$$metaType(src_l) \in Classes_{MetaModel} \tag{2.10}$$

$$metaType(v) \in Classes_{MetaModel} \Rightarrow \neg isAbstract_{metaType(v)} \tag{2.11}$$

$$c = metaType(src_l) \Rightarrow (metaProp(l) \in allProps(c)) \wedge \Big(metaType(trg_l) \in assignables(type_{metaProp(l)})\Big) \tag{2.12}$$

$$l_1 \neq l_2 \wedge metaProp(l_1) = metaProp(l_2) = p \wedge src_{l_1} = src_{l_2} \Rightarrow (\neg isOrdered_p \wedge index_{l_1} = index_{l_2} = 0) \vee (isOrdered_p \wedge index_{l_1} \neq index_{l_2}) \tag{2.13}$$

$$lower_p \leq |\{l | src_l = v \wedge metaProp(l) = p\}| \leq upper_p \tag{2.14}$$

$$v \notin allComponents(v) \tag{2.15}$$

$$l_1 \neq l_2 \wedge isComposite_{metaProp(l_1)} \wedge isComposite_{metaProp(l_2)} \wedge trg_{l_1} = trg_{l_2} \Rightarrow src_{l_1} = src_{l_2} \tag{2.16}$$

Only object values (instances of classes not primitive types) can be sources of links (2.10). This is because only classes can have properties, and primitive types do not relate values to each other. Only non abstract classes can be instantiated (2.11). This means there can be no object value that is the instance of an abstract class. The instantiated property of each link must be a property of the source value's class and the target value must be assignable (i.e. the type of the target value must fit the property's type) (2.12). Link indices are used to denote the ordering of property values (2.13). For non ordered properties all indices must be 0; for ordered ones, two values of the same property linked to the same source value must not have the same index. Multiplicity of properties must not be exceeded (2.14). Composition has the following semantics: composition defines a whole-part relationship. It must be acyclic: a container must not contain itself, neither directly, nor as part of a contained component (2.15), and a component can not be the component of two container (2.16).

Formal MOF-like formalism

Definition 23 (MOF-like meta-modelling formalism, *math.*)
A MOF-like meta-modelling formalism $MOF = (L, SM, SD)$ *is define by the three components language* L*, semantic mapping* SM*, and semantic domain* SD*. The language is the set of all meta-models:*

$$L_{MOF} =_{def} \{\mathfrak{M} | \mathfrak{M} \text{ is a meta-model}\}$$

The semantic domain is the set of all sets of models for all meta-models:

$$SD_{MOF} =_{def} \bigcup_{\mathfrak{M} \in L} \{M | MetaModel_M = \mathfrak{M}\}$$

Given a meta-model $\mathfrak{M}$ *I define the set of all instances of this meta-model* $models(\mathfrak{M})$*, i.e. the set of all models for this meta-model as*

$$models(\mathfrak{M}) =_{def} \{M | M \text{ is a model } \wedge MetaModel_M = \mathfrak{M}\}$$

This function $models$ *defines a mapping between* L *and* SD*:*

$$SM_{MOF} =_{def} models$$

2.3.3 Problems of Existing MOF-based Meta-Modelling Platforms

Meta-modelling platforms are about meta-modelling languages and frameworks. As with all languages and frameworks, there are typical questions are. Does the used syntax description language use the right level of abstraction? How convenient and save can the framework be used? Are there more effective languages and frameworks? In this section, I discuss problems of existing MOF-based meta-modelling platforms.

Missing Feature Refinement Semantics

I start my analysis with the meta-modelling languages. The newest MOF recommendations define two different meta-meta-models. The EMOF model provides a very minimalistic feature set, the CMOF a very extensive one. The EMOF concepts form a sub-set of the CMOF features, because the CMOF model is an extension, by means of object-oriented generalisation, of the EMOF model. This means, that the CMOF model provides additional means for the expression of meta-models. All CMOF constructs could also me described in EMOF (at least in conjunction with a language for well-

formedness rules), but of course this always requires manual implementations in a meta-model, where CMOF provides this functionality as part of the meta-modelling language.

There are two features in CMOF that may significantly enhance the convenience and safety in meta-modelling, when complex inheritance structures are important to the meta-model. These are called *feature redefinition* and *property sub-setting*. Both are refinement relations between the features of meta-model classes that restrict the semantics of a feature, e.g. limit the possible values for an attribute, or restrict the return type of an operation. General classifier-based refinement, known as specialisation or generalisation, becomes more powerful with the possibility to refine the classifier features in the context of a specialisation. Meta-modelling without feature refinement could be compared to object-oriented programming without the possibility to override methods in the context of class specialisation.

The UML 2.x meta-model [74, 73] extensively uses these new CMOF features. The UML meta-model uses this features to define the relationships between construct definitions. The UML defines general constructs in an abstraction library. The constructs in this library are specialised in different infrastructure modules, which are general enough to be used in multiple languages, for example for MOF itself and for UML. Within the UML superstructure, these infrastructure constructs are again specialised. And even within the UML superstructure constructs can be associated with different abstraction levels, know as the UML compliance levels 0 to 3, which provide different versions of UML based on level of language detail.

The static semantics of feature refinement is defined in the UML infrastructure recommendation. But these definitions only state what statically correct models are. Sub-setting for example requires that all the values in a sub-set property are also contained in the superset. With this definition it is possible to check an existing model. But this definition is not necessarily sufficient to describe what happens when a property is changed. Meta-modelling frameworks are dynamic modelling environments; they are about model changes. For example, when a new value is added to a property, what happens to the value sets of related, e.g. sub-setted, properties? UML is currently missing dynamic semantics for feature refinement that deals with modifying actions like setting, adding, or deleting values.

Weak Language Mappings

Meta-modelling frameworks require to map meta-modelling concepts to a programming language. The existing language mappings, especially for Java, like the mapping of EMF or JMI and JMI-like mappings don't exploit all

offered programming language features. This leads to programming that is not as type-safe as possible. This regards the change of property types in the context of inheritance (property redefinitions) and value collections with specific elements types.

Meta-model properties are mapped to pairs of *get* and *set*-methods in corresponding programming language interfaces. Existing platforms either do not allow any form of property type refinement in their meta-modelling language or they do not reflect type changes in the language mapping. Many programming languages, including Java, allow the change of method return types or method parameter types in the context of inheritance. Property type refinement, therefore could be mapped to such type changes in the programming language.

Properties with multiplicities higher than one are usually mapped to collections of values, using corresponding collection types in the programming language. Many programming languages allow collection types with type parameters, e.g. *generics* in Java. Such parameters allow type-safe access to such collections based on the type of the contained elements. Unfortunately, existing language mappings do not utilize this.

The missing use of these helpful programming language features leads into unnecessary cumbersome and not statically type-safe programming patterns. It requires heavy use of type-casts and manual type-checking, e.g. use of the *instanceof* operator in Java. For example: each collection access is accompanied with a type cast; when a collection contains several types of elements (lack of sub-setting use), collection access requires a switch based on the concrete element type and applications of the *instanceof* operator.[8]

Combination and Integration of Related Models

Especially in the modelling of languages, different models, covering different aspects of the same thing, have to be used with each other. This requires either the actual or at least logical combination of different models into a bigger model. This is an aspect that is often completely neglected by meta-modelling platforms and corresponding programming environments.

[8]One might object that with DSM of language tools, no code has to be written by hand and programming safety issues are irrelevant. But, as you will learn in the chapters on aspects representation and semantics, DSM of language tools gains much of its flexibility from the possibility to fall back to general purpose programming languages for language details that exceed the expressiveness of used meta-languages. It is unlikely that DSM of language tools is ever possible completely without programming; it is possible that there will always be programming parts of language tools manually, or manually changing generated language tool code.

Possible solutions include model transformations, described with model combination languages, or the use of specific *model merge* languages. The UML 2.x standard defines a generic model merge, that combines meta-model packages and their contents based on names. For example, two packages modelling different aspects of the same concepts can be merged into a package that includes these concepts with both aspects. In this process the concepts are identified with each other using their names as reference. Such methods are today only ideas with lack of proper formalisation and tool support.

Integration of Platform Objects

A meta-modelling platform usually defines a sealed model space; no support is given to combine a model of one platform with models or other entities in other platforms. Meta-modelling frameworks sometimes soften confinement, because they allow interactions between models and programming platform based on APIs and programs. When models are used to define tools, they have to be related to tool components written in the underlying programming language. It is therefore necessary to relate models with existing software components within the confined model space.

Even though meta-modelling platforms usually don't provide means for direct integration of other platforms, many meta-modelling frameworks introduce integration of programming language entities within models and meta-models. EMF for example allows to determine the concrete programming language class that realises a given meta-type. Such concepts are used in practice but are usually born out of pragmatism. Influences and effects of existing integration concepts aren't studied or formalized.

Other problems are behavioural and derived features. Meta-modelling platforms allow to define behaviour and derived features by interface declaration. They do not provide concepts for implementing those features. This problem is relayed to meta-modelling frameworks, which have to provide the means to provide feature implementations. Eventually, such behavioural and derived features are realised with program blocks written in the underlying programming language. The concepts that are used to connect feature declarations and their implementation are often crude and do not use the full potential of meta-modelling platform and underlying programming platform.

2.4 Related Work

The Java Metadata Interface (JMI) [75] presents a Java language mapping for the MOF 1.x recommendation, and thus gives Java users a meta-modelling

framework based on the MOF, which does include a Java language mapping itself. JMI is implemented, among others, by SUN's Metadata Repository [70] and the ModFact project [82].

The two mentioned MOF/JMI implementations MDR, and the ModFact repository, as well as the EMF meta-modelling platform, offer a common feature set and have all similar design. They offer a built-in repository component for meta-models, support code generation for user defined repository components based on meta-models. Models and meta-models can be loaded into repositories, using XMI-based [80] import and export facilities. Model elements can be accessed using either generated meta-model-dependent interfaces or meta-model-independent interfaces. All three implementations allow access to the model via classes, properties and methods that have names and types based on a corresponding meta-model.

The basics of these frameworks are well understood and proven. Therefore, I use the same architecture in my framework, as well does the language mapping follow the general rules for a MOF-based repository that are defined in JMI. However, the existing frameworks and their corresponding meta-modelling languages do not support the new MOF 2.x features: refinement of classifier features and sub-settings between properties. Basically, existing platforms cover the smaller brother of CMOF called EMOF (essential MOF), which is tailored for simpler and easier to realize modelling.

For the CMOF model, the MOF 2.x IDL [78], issued by the OMG, presents a MOF language mapping to OMG's platform-independent Interface Definition Language (IDL) [72]. Unfortunately this work is heavily constrained by limitations imposed through the very strict static type system of IDL. For instance, the type constraints implied by the redefinition of properties cannot be reflected in the target language, because IDL does not support operation overwriting with changing return types (covariant return types). This work presents value collections with concrete types. Since IDL does not have a built-in mechanism for generic types, those collections are generated for each type defined in a meta-model. This idea can be improved to work with generic collection types in Java, and can be enhanced to exploit the possibility of covariant return types offered by the Java programming language.

The implementation of associations is an often discussed aspect in modelling. Associations can be implemented as objects in their own right, which realize a bi-directional relationship using references from the association object (link) to the linked objects. This approach is chosen in MOF/JMI and was favoured by Rumbaugh et al. [91, 37]. The approach, championed by Graham et al. and used in EMF [39, 37], realizes associations ends as references that are constrained to be opposites of each other. I later use the second

	language	generics	qualifiers	sub-settings, redefines	pure reflection	IDE
A MOF 2 for Java	CMOF	yes	yes	yes	no	eclipse
MOFLON	CMOF	no	yes	yes	yes	Fujaba
MDR	MOF1.4	no	no	no	yes	netbeans
EMF	Ecore	yes	no	no	yes	eclipse

Table 2.1: A MOF 2 for Java in comparison with other MOF-like modelling frameworks based on framework features

approach, because it is easier to unify with property sub-setting, which is just another way of constraining properties. In [24] Diskin and Dingel discuss this issue from a mathematical perspective.

Ameluxen et al. [5] developed templates of Java code that realize the semantics of sub-setting in association ends and attributes or more general in properties. Here the update of a property value also triggers updates to subset properties in order to keep values of properties subsets of each other. This can be generalized: sub-setting is only a special kind of constraint for property values, another constraint kind is imposed by associations.

The feature refinement feature *sub-setting* and *derived unions* of the CMOF model were formalised by Alanen and Porres in[3] based on set-theory and first-order predicate logic. They give an operational semantics for the four basic operations: adding a property value, removing it, inserting a value into an unordered set, and inserting into ordered sets of property values. The semantics is given as pre- and post-conditions for these operations.

Alanna Zito et al. in [126] formalised the package merges that are performed to build the UML meta-model for different compliance levels based on merging different packages into different language configurations. They provide a mathematical package merge definition that ensures the compatibility of merged and original packages. This definition is more rigorous than the original OMG definition, which is more like an idea given in natural English than a actual precise definition.

When this thesis was written, several MOF-like meta-modelling frame-

works existed or were concurrently developed. In Fig. 2.1, I compare the most important of these frameworks based on shared features. The characteristics that distinguish the different frameworks from each other are: the used meta-modelling language, language mapping support for Java generics, support for UML qualifiers, sub-settings and redefinitions, whether the framework allows purely reflective programming without generating meta-model specific interfaces, and the IDE-platform that the framework targets.

The first row shows the framework *A MOF 2 for Java* [105, 102] that was developed for this thesis. The goal was to create a meta-modelling framework that resolves the limitations of MDR, which was at this point in time the only noteworthy meta-modelling framework. MDR [70] is based on the old MOF 1.4 recommendation and neither supports Java generics nor important MOF 2 features. MOFLON [6, 5] was developed at the same time than A MOF 2 for Java with very similar design goals. The *Eclipse Modeling Framework* (EMF) [117] was created with a completely opposite rational. The EMF developers wanted a meta-modelling framework with an even simpler meta-modelling language than MOF 1.4. While A MOF 2 for Java and MOFLON were intended for big, highly modularised meta-models with extensive reuse of language concepts, EMF is ideal for the small simple meta-models of DSL.

2.5 A New Meta-Modelling Platform Based on the CMOF-Model

2.5.1 Motivation

Some of the earlier described problems with existing meta-modelling platforms manifested during the development of a meta-model for SDL (see chapter 5). Especially problematic are missing feature refinements and weak language mappings in existing meta-modelling frameworks. To deal with SDL's complexity adequately, the meta-model has to contain deep inheritance hierarchies of meta-model classes. Modelling the features within these class hierarchies and programming with instances of these classes without proper feature refinement and corresponding language mappings was not bearable knowing that with CMOF at hand, everything could be modelled with more clarity, safety, and in a more compact meta-model. Similar problems arise in projects dealing with languages of similar complexity. The UML2 and UML2 Tools project with in the Model Development Tools (MDT) (developed within the eclipse community) [120], for example, realise a repository for UML based on EMF. The lack of support for feature refinement and an corresponding language mapping within EMF lead to a repository that is

stripped of many advantages of the UML meta-model: the different UML compliance levels cannot be used, most of the modularity is lost, abstraction hierarchies are melted down to the bare minimum realisable in EMF.

In this chapter, I describe possible solutions for the existing problems *missing feature refinement semantics* and *weak language mappings*. The solutions are accompanied with a discussion and comparison to corresponding related work. I realised the concepts presented in this chapter in a new meta-modelling framework based on the CMOF model: *A MOF 2 for Java*. This framework features a Java language mapping that is losely based on JMI, extended with concepts to utilise Java generics and type variations (needed for property redefinition). I present dynamic semantics for sub-setted and redefined properties, which ensures the statically given semantics explained in the MOF 2.x recommendations.

Furthermore and not further explained in this thesis: *a MOF 2 for Java* includes an implementation of a package merge mechanism to integrate different meta-models. The platform provides an abstract implementation layer that allows to provide implementations for behavioural and derived features in different and mixed implementation languages. It also allows to represent arbitrary Java values as model elements.

2.5.2 Language Mapping

A language mapping is needed to use MOF-based models in a programming language; it defines how model elements are represented by objects in the programming language, how such elements can be created, modified, or deleted using a programming language. A language mapping maps CMOF constructs to the constructs of a programming language. In case of the object-oriented CMOF-model and the object-oriented programming language Java, this is often straightforward: model elements are mapped to Java objects, MOF classifiers to Java interfaces and Java classes, CMOF properties to Java properties[9], and so on.

I choose to realize a CMOF mapping for the Java programming language, because Java offers a flexible type system. Compared with IDL (the recommended language to describe an interface to MOF), Java provides two important features that IDL does not: (1) *Covariant* return types that were introduced with Java Platform 2 Version 1.4 and (2) *generics*, introduced with Java Platform 2 Version 5.0. Those features will prove necessary for a convenient modelling API. The importance of covariant return types and generics will be discussed later in this section. For more information on

[9] Java properties are member variables, accessible through a pair of get- and set-methods

static type-checking and static semantics in object-oriented languages refer to [10, 38].

The previous MOF recommendation MOF 1.x [77] and its Java mapping defined in the *Java Metadata Interface(JMI)* [75], already constituted the common practices for a MOF to Java mapping. I propose a language mapping that follows these practices: Every model element is represented through a proxy object that implements the interface that corresponds to the corresponding meta-class. To call operations on a model element, or to access its attributes, or linked objects, corresponding Java methods have to be invoked. For more information about MOF semantics and language mappings refer to the old *MOF 1.4* [77] and *JMI* [75] or the new *MOF2.x Facility and Object Lifecycle* [81] and *MOF 2.x to OMG IDL mapping* [78] recommendations.

I extend this mapping to solve two problems: (1) JMI does not have support for CMOF's feature redefinitions, (2) JMI does not incorporate the possibility of generic types, hence programming with JMI often requires type-casts and reflection on runtime types. Common functionality, like model navigation, property updates, object creation, etc. has to be programmable in short and safe idioms. Following this rationale means that long chains of calls and numerous casts should not be necessary in the usual case.

The rest of this section will handle the detailed mapping of features; features are attributes, association ends, and operations. The next three subsections will cover (1) redefinitions of features, (2) merging of classes (a special application of feature redefinition), and (3) features with arbitrary multiplicity.

Mapping Redefinitions

The UML 2 infrastructure library (which includes abstract, basic definitions for the MOF model) allows an element to redefine other elements in the context of a generalisation relation. The abstract concept *redefinition* has different semantics and syntactic implications on its concrete realisations.

For properties it must hold that a redefining property is as least as restrictive as the redefined, or, in other words, the redefining property has to *be consistent with* the redefined property. In detail, the multiplicity of the redefining property must be included in the multiplicity of the redefined property, the redefining property must be derived when the redefined is, etc. The constraint that the type of a redefining property must be covariant (it *must conform*) to the type of the redefined property, is important for the language mapping. A property $p'{:}B$ of type B can redefine $p{:}A$ of type A, if B is a direct or indirect subtype of A, denoted as $A \triangleleft\!\!- B$. This is a *covariant*

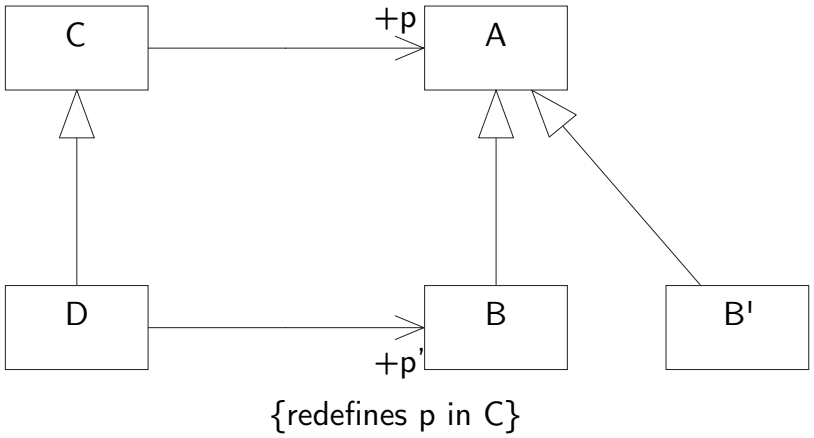


Figure 2.6: Redefinition example.

type change.

But for most implementation languages more restrictive rules apply. Because many of those languages try to assure static type safety, the type system has to be more restrictive. Some of these languages are Java, C++, and IDL. Most of those languages have the following, or even more restrictive redefinition semantics: In the context of an output-access, like getting a property value, or receiving the return of an operation call, the type can change *covariant* when the accessed feature is redefined. Since the redefinition can masquerade as the redefined, the redefining element can be accessed in any context the redefined can be accessed in. In the context of an input-access, like setting a property's value, or providing the arguments for an operation call, the type can change only *contra variant*; because only this way the redefinition can still masquerade as the redefined. In the example, p' can be access in any context that p is accessible. For properties that allow output- and input-access, like member variables, only *invariant* type changes are allowed.

I want to investigate what this diversity means for mapping the example given in Fig. 2.6. When a class C with property $p\colon A$ is specialized by class D with property $p'\colon B$, where p' redefines p and $A \triangleleft\!\!- B$, then according to the basic JMI mapping rules two interfaces are created. Interface `C` with methods `C::getP():A` and `C::setP(value:A):void`, as well as interface `D extends C` with methods `D::getP():B` and `D::setP(value:B):void` will be created on the Java side.

In Java terms: `D::getP():B` redefines `C::getP():A`, since it has the same signature. Fortunately Java supports covariant return types, and this piece of language mapping preserves the intended semantics. The set-methods, on

the other hand, are problematic. In order for a method to redefine another method (or *override*; to overwrite is what it means to redefine in Java), the arguments of the redefining method must only change contra variantly. But the argument of the update method changes covariantly. Therefore, `D::setP(B value):void` does only *overload* `C::setP(value:A):void`. Unfortunately overloading does not have the semantics wanted.

Take the call `c.setP(aValue)`, where `c` is a reference of type C and `aValue` is of type B' with $A \triangleleft\!\!- B'$ but not necessarily $B \triangleleft\!\!- B'$. Since references are polymorph in Java, it is possible that `c` references a value of type D. What are the semantics of this call in this particular context?

The proxy object that implements the interface `D` has to implement both methods `C::setP(value:A):void` and `D::setP(value:B):void`. Since the two methods simply overload each other (despite their shared name), the mentioned call `c.setP(aValue)` will invoke `C::setP(value:A):void`. This is because `aValue` has type B', which is incompatible with B.

But this is not what MOF intends; instead I want $p' : B$ in type D to be updated, since it redefines $p : A$ in C. There is only one slot (value container) defined by the redefining property p'. This slot is used for both properties p and p' because both describe actually the same property just with different types. The reason for the mappings failure is that you try to do a covariant type change on both, a output-access (`getP`) and a input-access (`setP`), on the same property. But static type safety can only be assured for a writing access with contra variant type change. In other words static type safety for redefining a property with both output- and input-access can only be assured for *invariant* type changes.

The solution to this problem is to postpone some type-checking from compile time to run time. The proxy object's implementation of method `C::setP(value:A):void` must realize that it is redefined; it has to check whether `value` is of type B or not. If `value` is of type B, it delegates and calls `D::setP(value:B):void`; if it is not of type B, it raises a type-check exception. That way, retrieving a value can be type-checked statically. The type-checking of setting a value is only possible against the most general type of a redefinition; complete type-checking, however, is possible at runtime.

Redefinitions, where the redefining element has a different name, are another troubling point. The mapping leads to Java methods with different names and hence different signatures, and therefore there is no redefinition at Java level. You can apply the same solution to solve this problem: The corresponding method implementation realizes that the represented property has been redefined, it does the needed type-checks, and delegates the call to the method that represents the redefining property.

For operations, MOF allows operation arguments to change covariant

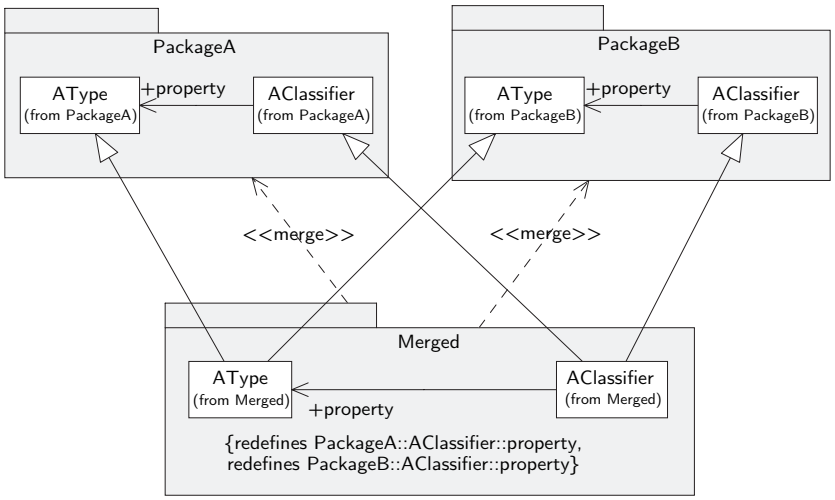


Figure 2.7: Merge example.

when redefined. This is a bit peculiar and the semantics are not further described. When operations with covariant arguments are mapped to Java, overloading semantics apply. Since overloaded methods are selected at compile time it cannot be assured that the wanted operation is called. As before, the implementations of the corresponding Java methods have to decide at runtime with operation is to be invoked. [10] [11]

Multiple Inheritance and Merging

Even more challenging for the Java type system is the redefinition of multiple properties as they commonly occur when packages are merged. In the example, shown in Fig. 2.7, two classes with a property of same name, but different types, are merged. For instance, MOF used this several times for the definition of the CMOF model. For example, the CMOF package is a merge of EMOF and UML infrastructure's Constructs.

In the example, `PackageA::AClassifier` and `PackageB::AClassifier` are merged into the class `Merged::AClassifier`. Based on the presented Java mapping, classes are mapped to Java interfaces and generalisation between meta-model classes to inheritance between Java interfaces. The merge

[10] This thesis does not explain the runtime semantics of operations in detail. But, CMOF operations can basically be mapped on Java methods with implementations that the user has to provide, since the operation behaviour cannot be expressed with CMOF.

[11] Possible other semantics for operation parameter that could be worked into a language mapping, are *multi-methods* [33]. The implementation of such operations is resolved based on the runtime types of the arguments of an operation call.

leads to multiple interface inheritance. The interface for `Merge::AClassifier` inherits from the corresponding interfaces of both super classes. The interface for `Merge::AClassifier` further more overrides two methods with the same name. These are the getter-methods for the property `property`, which is defined for both specialised classes `PackageA::AClassifier` and `PackageB::AClassifier`. The Java language does not forbid such a multiple override per se, but it does so, when the two overwritten methods have incompatible return types. And since `PackageA::AType` and `PackageB::AType` are unrelated and therefore incompatible with each other, this mapping leads to faulty Java code. Java even forbids this when the overwriting method has a return type, which is constructed to be compatible with both `PackageA::AType` and `PackageB::AType`. Although, `Merged::AType` specialised both `PackageA::AType` and `PackageB::AType`, the result is still invalid Java code.

The fact that such a mapping does not work in Java, because Java applies an unnecessary strict type system, does not necessarily mean that it cannot work for other static typed languages. From a type-checking perspective the overwriting method has a type that is covariant to the type of both overwritten methods.

For the Java mapping the only satisfying solution that I could find is to use the *combine* semantics described in MOF 2.x Core. Combine is a special kind of merge, where the merging package contains all classes and features as it would with the regular merge, but all redefinitions and generalisation relations to the merged packages are omitted. This leads to a package with types that have all desired features, but that do not conform with the types of the merged packages.

Higher Multiplicity

Properties can have arbitrary multiplicity; it means that they can represent a collection of values. Elements with such a multiplicity have to be mapped differently. For properties with higher multiplicity only one access method is generated with a Java type that allows to contain a collection of values. To update the values of the property, the collection retrieved through access method invocation has to be changed. In order to allow type safety and convenient programming, this collection has to preserve the type of its elements. Java supports generics to parameterise collection types with concrete element types.

Those collections preserve type safety for both, output- and input-access to its elements. But this causes another problem: Two collection types $Set<A>$ and $Set<B>$ are not compatible to each other, even if $A \leftarrow B$. That

means, if property $p' : B$ in D redefines $p : A$ in C and both property's have multiplicity that implies a collection of values then the mapping to `C::getP():Set<A>` and `D::getP():Set<B>` does not work, because the two return types are incompatible and would cause a compile time error. Generics are only compatible for invariant context type parameter due to the fact that a generic may use a context type parameter for an output- and an input-access.

For such cases Java allows to weaken the generics' context parameter: The reference type `Set<? extends B>` is covariant to `Set<? extends A>`, if `B` is covariant to `A`. The context parameter is now unbound to covariant types. The negative side of an unbounded context parameter is that every method that uses this context parameter as a type for one of its arguments (input-access), can not be called anymore. The reason is that the object beyond the reference of type `Set<? extends A>` could have a more restrictive context and therefore static type safety can not be assured anymore.

The solution to modify collections with unbound element type is the same as the solution for modifying redefined properties: The type-checking is postponed until runtime. I implemented generic collection classes that use the most general Java type `Object` as element type for any method that modifies the collection, but a concrete type for all methods that return elements of a collection. That way, static type safety is assured for output-access on the collection' members, but modifying access can only be type-checked dynamically.

2.5.3 Semantics for Associations and Property Sub-setting

The semantics for associations and property sub-setting in the CMOF model are explained in the MOF 2.x Core [79] and the related recommendation UML 2.x infrastructure [73]. However, the models in those recommendations are only described as static constructs, and therefore their semantics is defined only for static models; the recommendations express semantics only in static conditions. But model repositories—basically dynamic programs—define operations to create model elements, operations to add values to the properties of an object, or remove values from the properties of an object.

When implementing such operations, non-trivial questions arise that can not be answered by static constraints. In a repository, the values of sub-setted properties have to form sub-sets at all times. The basic problem is: when the user makes a local change to a model, the user would cause property values to not form sub-sets anymore. Therefore, the values of other properties have

to be changed automatically to satisfy all sub-settings.

I refine the semantics for associations and property sub-setting in this section. I propose a mechanism that uniformly includes semantics for associations and property sub-setting. This is reasonable, because associations and property sub-settings are two special cases for constraints between properties. Associations impose a bi-directional relationship between properties—namingly association ends; and property subsets imply that the values of one property always form a subset for the values of another property. Before I proceed to explain the semantics in a formal manner, I want to discuss my and other approaches to both subjects: associations and property sub-setting.

Associations have been used since the early works on object-oriented design; semantics and implementation strategies have been discussed widely. There are two general possibilities: (1) associations are classifiers in their own right with possible properties and specializations between associations; realized with association instances that link associated objects (Rumbaugh [91], MOF/JMI [79, 77, 75]); (2) associations are simply realized by unidirectional references with values that are constrained by the association (Graham, Bischof, Henderson-Sellars [39], EMF [117]). I used opportunity (2), because it defines an association as a constraint on the values of two corresponding properties. In MOF those properties are called association ends, which are basically properties of the association[12]). This allows me to handle associations in a similar way as I handle property subsets. However, realizing associations using constrained properties internally, does not mean that it is impossible to maintain the appearance of associations as classifiers and links as their instances.

Property sub-setting is a new feature in MOF and available in the CMOF-model. Amelunxen et al. [5] use the following approach: When adding a value to a property, it is also added to all subset properties and their subset properties and so on. This is straight forward; the values of properties remain subsets. When removing a value from a subset, remove it also from properties that are subsets to that property, concluding subsets remain subsets. I agree with the adding process, but the removing seems troubling. Imagine, adding a value to property p with subset property p' and deleting the same value from p again. It will be added to p', but not removed from p'. The remove operation does not completely reverse the add operation. This does contradict typical add/remove semantics.

[12]The CMOF itself (the same for the UML or older MOF version) use the Rumbaugh way of modelling associations. In CMOF an association is a classifier. As each classifier, an association has member properties, but in addition an association has member ends, sub-setting member properties. These special properties associate classifiers (the property' types) with each other

I propose a different approach: every time a value is added, you keep the information on which places the value is added (this information is later on called *update graph*). Whenever this value is removed, no matter on which of the properties it was once added to, it is also removed from all other properties. That way, you completely reverse the original intention of adding the value.

In the remainder of this section, I define the semantics for adding and removing property values on a set theoretic basis. After I give some basic definitions, that relate relevant CMOF constructs to sets and relations, I introduce the notion of slots, and sets of depending slots. Finally, I use those definitions to define semantics for adding values and removing values from object properties.

Properties, Slots, and Depending Slots

May P denote the set of all properties in a meta-model. I define for $p_k, p_l \in P$ the relation $p_k <_r p_l \Leftrightarrow p_k$ redefines p_l. It reflects redefinitions given in the meta-model. I define $p_k \ll_r p_l \Leftrightarrow \exists p_1, \ldots, p_n : p_k <_r p_1 \wedge \cdots \wedge p_n <_r p_l$ and the reflexive, transitive closure $p_k =_r p_l \Leftrightarrow p_k \ll_r p_l \vee p_l \ll_r p_k \vee p_l = p_k$. The equivalence classes of $=_r$ form slots. I define $slot(p) = \{p_i | p_i =_r p\}$ as a function that gives the corresponding slot to a property.

Later on I will use slots as container for property values. Because there is only one slot for all properties that redefine each other, the notion of slots has also redefinition semantics attached to it.

There are two relationships between slots, imposed by relationships between corresponding properties. Firstly, I define a relation for associations; for two slots s_1, s_2, $\rightarrow_a$ is defined as:

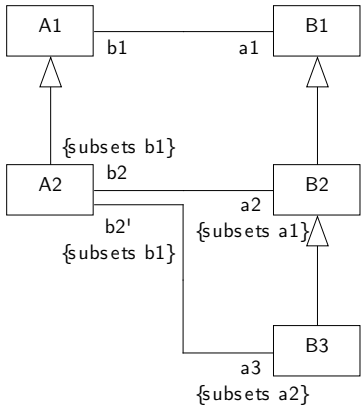
$$\begin{aligned} s_1 \rightarrow_a s_2 \quad &\Leftrightarrow \exists p_1 \in s_1, p_2 \in s_2 : \\ &\quad p_1, p_2 \text{ are the ends of an association} \end{aligned}$$

Secondly, the definition of a relation for sub-setting $\rightarrow_s$ is defined for two slots s_1, s_2 as:

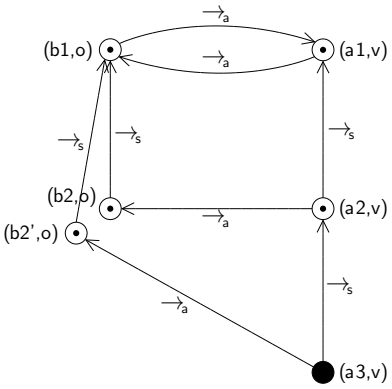
$$s_1 \rightarrow_s s_2 \Leftrightarrow \exists p_1 \in s_1, p_2 \in s_2 : p_1 \text{ subsets } p_2$$

Based on those relations, I construct a set of depending slots. Therefore, I define an update as $u = (o, v, s)$, where o denotes an object that contains slot s and v is the value that the slot s shall be updated (either added to the slot, or removed from the slot) with. To keep this simple, I only consider object values here. For that matter v denotes always an object. I define a set of depending slots regarding an update as a set of updates $ds(u)$. It is

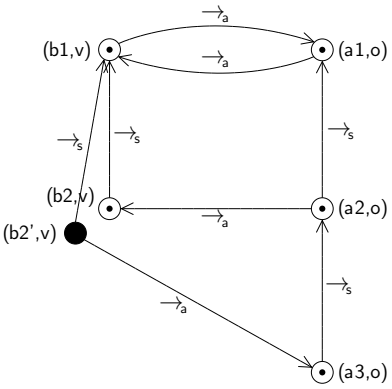
Model)



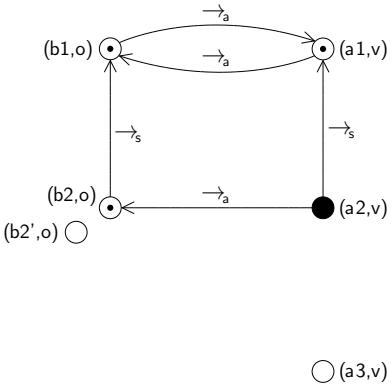
1) u=(o:A2,v:B3,a3)



2) u=(o:B3,v:A2,b2')



3) u=(o:A2,v:B2,a2)



4) u=(o:A1,v:B1,a1)

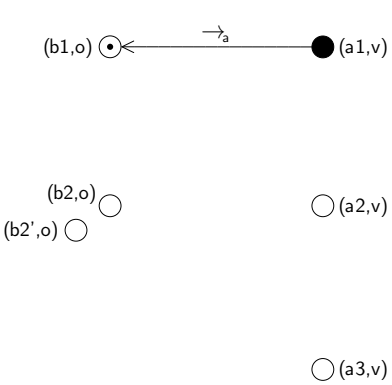


Figure 2.8: Update-set examples.

defined recursively; it contains the update u and all updates that are implied by associations and sub-settings for the properties in updates in $ds(u)$:

$$start: \quad (o, v, s) \in ds(u)$$

$$association: \quad \frac{(o_1, v_1, s_1) \in ds(u) \wedge s_1 \rightarrow_a s_2}{(v_1, o_1, s_2) \in ds(u)}$$

$$subset: \quad \frac{(o_1, v_1, s_1) \in ds(u) \wedge s_1 \rightarrow_s s_2}{(o_1, v_1, s_2) \in ds(u)}$$

Given the constrains in the UML infrastructure (that CMOF is based on), which regard validity of redefinition and subset contexts, $ds(u)$ can only contain $(o_i, v_i, s_i), (o_j, v_j, s_j) \in ds(u)$ with $(o_i = o_j \wedge v_i = v_j) \vee (o_i = v_j \wedge v_i = o_j)$. In other words an update will only concern two objects, the object that is updated and the value that the object is updated with.

The example in Fig. 2.8 shows a model with five properties with different association and subset constraints between them. Below the model you see different examples of update sets. Dependencies between all slots are drawn as arrows; every dependent slots is marked as $\odot$; the originating slot that the update is initiated on, is marked as big bullet. The first example (1) shows the update set for value v (an object that has at least type $B3$) on object o (at least of type A2) for property $a3$. Sometimes there are several reasons why a slot depends on the originating slot; they have different arrows pointing at them. Because of the symmetry of $\rightarrow_a$ the update set for adding a value v (at least type $A2$ on object o (at least of type B) to property $b2'$ in example (2) contains the same slots as in example (1). Examples (3) and (4) show smaller updates where the properties $a2$ and $a1$ are updated.

Add and Remove Values to Properties

Meta-modelling frameworks use operations on models to modify them. These can be object creation, deletion, or the adding and removing of values to and from properties. May Op be a set of such operations. I only want to give semantics for $\texttt{add}(o, v, p) \in Op$ and $\texttt{remove}(o, v, p) \in Op$. These are parameterised operations that add or remove a value v in property p of object o. May E denote the state of an model. $E = op_1 \circ \ldots \circ op_n$ results from successively calling operations $op_i \in Op$. For each E and object o, $f_{o,E}(p)$ denotes the set of values that the property p of object o has in state E. I use these function in predicates $P(f_{o_1,E}, \ldots, f_{o_n,E})$. Alternatively, I write $E \models P(f_{o_i}, \ldots, f_{o_n})$ to denote that a state E yields predicate P for the objects, properties, and their values in E.

The predicates $AllSet_{o,v,p}$ and $NonSet_{o,v,p}$ describe a model state, where all depending slots of update (o, u, p) contain the corresponding value (there are all set), and a model state where all those slots do not contain those values (there is non of these values set).

$$\begin{aligned} AllSet_{o,v,p} &:= \forall (o_i, v_i, s_i) \in ds\left((o, v, slot\left(p\right)\right)) : \\ &\quad \forall p_i \in s_i : v_i \in f_{o_i}(p_i) \\ NonSet_{o,v,p} &:= \forall (o_i, v_i, s_i) \in ds\left((o, v, slot\left(p\right)\right)) : \\ &\quad \forall p_i \in s_i : v_i \notin f_{o_i}(p_i) \end{aligned}$$

Using those predicates, the following rules describe `add` and `remove`:

$$\frac{E \models NonSet_{o,v,p}}{E \circ \texttt{add}(o, v, p) \models AllSet_{o,v,p}}$$

$$\frac{E \models NonSet_{o,v,p} \wedge E \circ op_1 \ldots op_n \models NonSet_{o,v,p}}{E \circ \texttt{add}(o, v, p) \circ op_1 \ldots op_n \circ \texttt{remove}(o, v, p) \models NonSet_{o,v,p}}$$

When adding values to a property, all depending properties are updated too. This is straightforward, opposite association ends are assigned accordingly and subset properties are updated to remain supersets. The remove, however, might be more peculiar. When removing a value, it is also deleted from the set of depending slots that was originally used to add the value. This especially means that for all $p_1, \ldots, p_n$ and $(o_i, v_i, s_i), (o_j, v_j, s_j)$ with $p_i \in s_i, p_j \in s_j$ and $s_i, s_j \in ds((o, v, slot(p)))$ and for all arbitrary predicates A:

$$\frac{E \circ \texttt{add}(o, v, p) \circ \texttt{remove}(o_i, v_i, p_i) \models A}{E \circ \texttt{add}(o, v, p) \circ \texttt{remove}(o_j, v_j, p_j) \models A}$$

In other words, no matter on what slot of depending slots of (o, v, p) I remove a previously added value from, all slots that were originally updated when adding the value, are considered. That way, all values are removed that are assigned to the original intention of adding a value. Example 1 in Fig. 2.8 for example: No matter whether o is removed from $b1$, $b2$, or $b2'$ or v is removed from $a1$, $a2$, or $a3$, they are all removed, because the reason for each value is in each slot that v has been added to $a3$. The same holds for the other examples.

The rules given here are far from complete. Non-trivial cases arise especially in the context of certain property features, like ordered values, uniqueness of values, multiplicities, non object-values, etc.

2.6 Conclusions

With the *A MOF 2 for Java* framework, you have a meta-modelling platform at hand which realises CMOF's feature refinement abilities, namely property redefinitions and sub-settings. Because complex meta-models, like UML 2.x's meta-model and the SDL meta-model presented in chapter 5, use these features heavily, my meta-modelling framework is a good choice to develop language tools for those meta-models. Beyond feature refinement, you can also profit from CMOF's advanced packaging constructs and merging abilities. This allows, for example, to deal with the UML meta-model as it is recommended in the UML 2.x recommendations without alteration and right away. The framework can be used to create UML repositories for the different compliance levels. There is no need to adopt the UML meta-model to a weaker meta-modelling language, such as EMOF or EMF's Ecore, as it was necessary in the UML2 eclipse project [120]. Of course, the presented results are also valuable for other meta-models of high complexity and for the development of future meta-modelling languages that may succeed the CMOF model.

Another important use-case for a meta-modelling language with feature refinement is the modelling of libraries for abstract language constructs. Such libraries define constructs based on abstract classes. The construct classes are later specialised. To allow the use of such abstract classes in as many concrete contexts as possible, high flexibility in refining class features is necessary. The idea of language construct libraries is used in the UML 2.x infrastructure [73], where the *abstraction* packages present a library of small cohesive language constructs, which are (re-)used throughout the different MOF and UML meta-models. I adopted the idea of construct libraries and re-used the UML infrastructure library for the SDL meta-model. I created more language construct libraries based on UML's abstraction libraries and used all libraries to define SDL's constructs themselves. Language construct libraries is an ongoing research matter. Concrete research puts emphasis on different ways to combine abstract and concrete language constructs. Examples for this research are [67], which uses aspect weaving to combine language constructs, and [43], which uses composition to inject general constructs into existing languages.

Despite the presented advantages of meta-modelling platforms with feature refinement, the industry so far mainly adopted meta-modelling technology based on simpler meta-modelling languages. The most commonly used MOF-like meta-modelling platform is eclipse's EMF. Even though the realisation of UML for example required an adopted meta-model and further programming efforts to create a UML repository based on EMF, the UML2

project is successful and used for many UML-based modelling tools. But EMF's major success is probably due to its use for developing DSLs. EMF's simplicity is praised by practitioners who state that they can develop DSLs more efficiently with EMF than with other technologies. On first sight, DSLs can not profit from more complex meta-modelling languages and things like feature refinement. Although there are good reasons, especially the use of abstract construct libraries to create DSLs, CMOF-based frameworks are experienced as overwhelming complex by industry engineers. Another aspect that circumvent the dissemination of CMOF-based frameworks is the huge amount of meta-languages and meta-tools that are available for EMF. This tool support gives EMF a leverage that is hard to overcome by competing technologies.

Chapter 3

Representation – Describing Textual Notations

3.1 Introduction

I showed in the previous chapter that models (language instances) are instances of meta-models (language descriptions) according to a meta-modelling formalism. I showed that models can be maintained in computer memory based on meta-model specific data-structures. But neither models as abstract mathematical elements, nor models as a bunch of objects in computer memory are usable for human beings. Humans need tangible objects. Humans need model representations that use concrete symbols to depict the abstract elements of a model. Examples for such representations are program-code or diagrams.

This chapter is about textual model representations. These are representations that use characters and words to represent models. In other words, representations are instances of textual languages. Similar to meta-models that I use to define sets of models, you can use notations to define the set of valid representations for the models of a language. Because textual representations are instances of textual languages, context-free grammars provide a meta-modelling formalism to define textual notations.

To describe textual notations, you need to map from textual languages described in a grammar-based meta-modelling formalism to languages described in a MOF-like meta-modelling formalism. The interesting questions that I want to answer are: what is the relationship between grammars and meta-models and how can this be used to define notations? How can you use notations, how can you create a model from its textual representation and vice versa? How can you facilitate knowledge about notations to enhance

representation related tools like model editors?

I start this chapter with general definitions for representations, notations, and some of their properties in section 3.2. The next section 3.3 turns towards textual representations, where I introduce textual notations on an abstract level, discuss their applications, and existing problems. Section 3.4 discusses related work. The following sections address some of the given problems. I provide a formal analysis of the relationships between context-free grammars and meta-models in section 3.5. In section 3.6, I present a framework to create meta-models from existing grammar-based notations. Finally, section 3.7 presents meta-language to describe textual notations for MOF-like meta-model-based languages, and it presents a corresponding meta-tool to automatically create textual model editors.

3.2 Model Representations and Notations in General

In this section, I briefly introduce some formal definitions of notations, representations, and their properties based on the formalism notion introduced in 2.2.1.

Definition 24 (representation) *A* representation *represents a model in a human comprehensible way. If a representation only represents a part of a model, it is called (*partial representation*). If a representation does not represent a model in all its detail, it is called an (*abstract representation*). If a representation represents more than one model of the same language, it is called an (*ambiguous representation*).*

A representation has to be well defined. That means there has to be a formalism that defines what a representation is and what not.

Definition 25 (notation) *A formalism that defines representations and their relationships to language instances is called a* notation*. A notation N is a formalism $N = (R, \psi, \wp(M))$. R is a set of possible representations; ψ a function that assigns each representation a set of represented models; M is the set of all models in a language, $\wp(M)$ denotes the set of all sets of models. I also write $n\psi m$ iff $m \in \psi(n)$. If $m\psi n$, I say n* represents *m.*

The notation thereby describes a set of representations. Notation relates to representation like language relates to language instance. Notation N is *unique* when $\forall r_1, r_2 \in R : r_1\psi m \wedge r_2\psi m \Rightarrow r_1 = r_2$, and it is *unambiguous* when $r\psi m_1 \wedge r\psi m_2 \Rightarrow m_1 = m_2$. At first sight, unambiguous notations

seem to be useless, since its representations fail to denote one model. Lets look at the following example: you have a MOF meta-model consisting of two classes. You use UML class diagrams as a MOF notation: you have one UML class diagram that shows you one of these classes. This class diagram is a representation for the MOF model with the two classes. It is obviously a partial representation. This diagram also represents the MOF model that consists only of the class that you see in the diagram. It also represents indefinitely many MOF models: all models that contain the class showed in the diagram. You need to know more about the semantics of a language to determine whether ambiguous notations are reasonable or not. Therefore, I extend the normal formalism concept.

Definition 26 (formalism with abstractions) *A* formalism with abstractions $F = (M, \psi, D, \succeq)$ *consists of a language* M*, which is a set of models; a semantic mapping* $\psi : M \to D$ *that assigns each model an element of the semantic domain* D*, and abstractions* $\succeq$*.*

The relation $\succeq \subseteq D \times D$ *describes a reflexive half-order on* D *which reflects the possible* abstractions *within the formalism's semantic domain. Furthermore, I use the symbol of the abstraction relation for models:* $m \succeq m' =_{def} \psi(m) \succeq \psi(m')$*. Two element of the semantic domain* $d \in D$ *and* $d' \in D$ *are called* semantically equivalent *if* $d \succeq d'$ *and* $d' \succeq d$*.*

In the example, I used the MOF meta-modelling formalism $MOF = (MM, \mu, \wp(M), \supseteq)$ where MM is the set of all meta-models, μ is a total function assigning each meta-model the set consisting of its instances, $\wp(M)$ denotes all possible sets of instances. Abstraction is given by the normal $\supseteq$ relation: the instance set of a more abstract meta-model contains all the instances of the more specific meta-model.

A notation is *consistent* when for all $r \in R$ the minimum $min_{\succeq}(\psi(r))$ exists. Each unambiguous representation function is also consistent, because the set of represented models contains only one model for each representation.

Definition 27 (representative formalism) *For each consistent notation, I define* $\psi^*(r) = min_{\succeq}(\psi(r))$*. For each formalism* F *and notation* N*, I can define a* representative formalism $F_N = (R, \psi_N, D, \succeq)$ *where* $\psi_N(r) = \psi(\psi^*(r))$*.*

This representative formalism has the same semantics but uses a different language. You can use a consistent notation to find the most abstract model for a model representation.

Definition 28 (reverse notation) *For a formalism $F = (M, \psi, D, \succeq)$, a consistent notation $N = (R, \psi, \wp(M))$, and a representative formalism $F_N = (R, \psi_N, D, \succeq)$, there might exist a notation $N^{-1} = (M, \psi^{-1}, \wp(R),)$ for F_N. If N^{-1} is consistent and $\forall m, m' \in M; r \in R : \psi^*(r) = m \wedge \psi^{-1^*}(m') = r \Rightarrow m \succeq m' \wedge m' \succeq m$, N^{-1} is called a* reverse notation *for F_N.*

You can use a reverse notation to create the most abstract representations for a model. Creating a model from a representation and creating a representation for a model are the two major applications for notations.

3.3 Textual Representations

In the last section, I described the abstract concepts notations and representations. Now, I introduce formalisms to define concrete notations. These are the formalisms that you use to determine the possible sets of representations and therefore determine the kind of notation. From here on, I am focusing on textual notations. I discuss formalisms for textual notations in the next part, than I present applications for textual notations, and discuss the problems that arise, when you realise these applications.

3.3.1 Formalisms for Textual Notations

I use *context-free grammars* as a meta-modelling formalism to define the possible sets of representations for a notation. There are two formalisms for context-free grammars. These formalisms use the same language for grammars, but use different semantics. The first formalism describes languages as sets of *sentences*. The semantics assigns each grammar a set of *sentences*. These sentences are the instances of the textual language described by the grammar. These are the sentences that can be derived from the grammar rules. The second formalism uses sets of *parse trees* as semantic domain. These parse trees are the syntaxes of the instances of the described textual language. A parse tree (as a syntax) describes how a sentence can be derived from the grammar rules. Grammars therefore describe both a set of sentences and a set of parse trees, and there is a surjective mapping from the set of parse trees to the set of sentences.

Since the expressive power of grammars is limited, you might want to narrow the resulting sets of representations. Therefore, I extend grammar formalisms with an additional formalism for *constraints*. I need to define special formalisms that allows to restrain the languages defined in a given meta-modelling formalism. Let $F = (\mathfrak{L}, \psi, D = \{L_1, \ldots, L_n\})$ be a meta-modelling formalism with $\mathfrak{L}$ as the set of possible meta-models and possible

languages L_i. You can apply a *constraining meta-modelling formalism* to further constrain a language $\mathfrak{L}_\mathfrak{i}$ defined with F.

Definition 29 (constraining formalism) *A* constraining meta-modelling formalism *is a formalism* $F(M) = (\mathfrak{L}_{F(M)}, \psi_{F(M)}, D_{F(M)})$ *where the parameter* $M \in \mathfrak{L}$ *is a meta-model in* F *and* $\forall M' \in L_{F(M)} : \psi_{F(M)}(M') \subseteq \psi(M)$.

The language of the constraining formalism depends on the meta-model that it constrains, because the constraints that you want to formulate use, for example, names and symbols from the constrained meta-model. The semantic domain of the constraining language are languages that are sub-sets of the language originally defined by the constrained meta-model, because constraints only constrain the original language. You can combine formalisms F and $F(M)$ to $F \circ F(M) = (L_\circ, \psi_\circ, D_\circ)$, where

$$\begin{aligned} L_\circ &= \{(M, M') | M \in \mathfrak{L} \wedge M' \in \mathfrak{L}_{F(M)}\} \\ D_\circ &= \{L | \exists M \in \mathfrak{L} \wedge L \subseteq \psi(M)\} \\ \psi_\circ((M, M')) &= \psi_{F(M)}(M') \end{aligned}$$

The language of the constrained formalism consists of tuples: the first part constitutes the meta-model, the other part comprises constraints for the meta-model. The domain of the constrained formalism is the domain of the original formalism plus all subsets of the original domain elements.

3.3.2 Applications for Textual Notations

Models cannot be exchanged. Only their representations can be communicated. Since you are interested in computer languages, the main concern is to transport models from the human mind into computer memory and back again. Notations allow humans to express language instances in a way that allows computer programs to create a copy of that language instance in memory. On the other hand notations allow computers programs to create a human comprehensible representations for stored instances. This communication is realised with specific tools: parsers, pretty printers, and editors.[1]

Using Representations to Create Models: Parsing

I will briefly introduce the parsing process, its distinct elements, the involved formalisms and the tools that have to be created to realise parsing.

[1] Of course notations can also be used to communicate language instances between humans or to exchange language instances between computers programs. Communication media are: humans use pen and paper, computer programs using technical protocols, networks, and files.

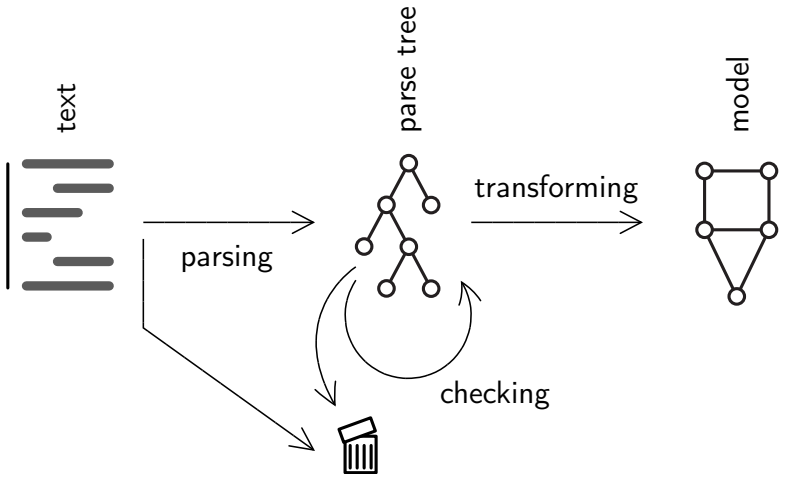


Figure 3.1: The different steps involved in creating a model from text.

Parsing provides a transition form a textual representation as sentence (notation instance) to representation as parse tree (notation instance syntax). Within a grammar-based meta-modelling framework, you can create a *parser*, which is a computer program, written or generated for a specific grammar, that takes text as input and produces either a parse tree or an error from that input. Using existing frameworks like [90], allows to automatically generate such parsers based on a given context-free grammar.

A parse tree can be *checked* for a set of semantic constraints. Within a grammar-based meta-modelling framework that supports a constraining formalism, you can create a *checker*, which is a computer program, written or generated for specific semantic constraints. A checker thereby uses the semantics of the constraining formalism. A checker either *accepts* a parse tree or produces an error.

You can create a language instance from a parse tree. Combining grammar-based meta-modelling framework and MOF-like meta-modelling framework, you can write a *transformation*, which, in this context, is a computer program that creates a language instance (model) from a parse tree. Such a program traverses the parse tree and creates corresponding construct instances (model elements) for the visited parse tree nodes. This computer program reflects the semantics of the notation.

Fig.3.1 pictures the following process. Lets assume that you have a notation with a set of representations defined in an constrained grammar-based meta-modelling formalism. This notation uses a grammar and a set of semantic constraints to define the set of valid representations. You have some text provided by the user and now want to create a corresponding language

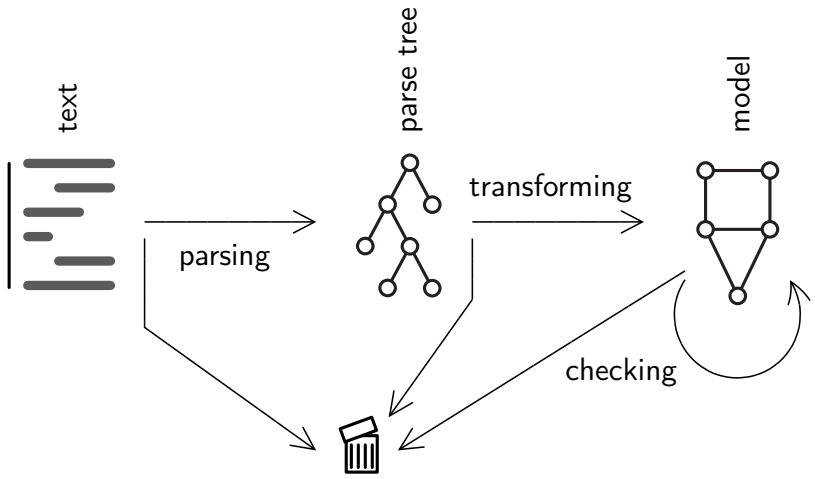


Figure 3.2: The different steps involved in creating a model from text with delayed semantic checks.

instance. Firstly, you use a parser generated from the notations grammar. It provides you with a parse tree for the user text or produces an error. Then you can use a checker to check the parse tree. This either accepts the user input or produces an error. After that, you use a computer program to create a language instance from the parse tree according to the notation's semantics.

Another possible process is shown in 3.2. Since you often also use constraining formalisms on MOF-based meta-models to achieve the same semantic constraints on models that are already covered by the notation, you can delay checking and do it only on the model. The only semantic constraints that you have to check on the parse tree, are those semantic constraints that are due to differences in the expressive power of grammar and MOF-like formalisms. These constraints are implicitly existing within the notations semantics. The notation semantic mapping is a total mapping for the constraint set of representations and conclusively a partial mapping for the unconstrained set of representations. The resulting transformation based on the notation semantic mapping causes an error for all representations (parse trees) that it does not map.

Using Representations to Present a Model: Pretty Printing and Annotations

Based on a reverse notation, pretty printing describes a reverse parsing process. Combining MOF-like and grammar-based meta-modelling frameworks, you can write a computer program that creates a parse tree from a model

according to the semantics of a reverse notation. Within a grammar-based framework, you can write or generate a *pretty printer*, which is a computer program that prints a text from a parse tree. This printer simply traverses the parse tree and prints text for each visited node according to the grammar rules.

When a model is processed by a modelling tool, e.g. an analyser or simulator, it might be desirable that the output of these tools is presented as *annotations* to a representation of the model. Annotations, for example, can be displayed as red under-lines, symbols attached to a line of text, highlighting of text, etc. Annotations have to be anchored in a representation. For textual representations, such an anchor can be realised as a position or range in the text. Based on a textual notation, you need to create an *annotator* within a grammar-based framework and a MOF-like modelling framework. An annotator is a computer program that finds positions and ranges in a textual representation based on an input model element. An annotator uses a reverse notation to determine elements within a parse tree from a model element, and uses the grammar formalisms semantics to determine text position and range for elements in a parse tree.

Using Representations to Edit a Model: Editors

Editors combine parsing and pretty printing: the user types text, the text is converted into a model. This is repeated, should the user change the text. When the model changes, or there is no text available for the model, the model is pretty printed. Editors allow continuous user changes, model changes, and transformations between representation and model. In contrast to parsing, text that is not a valid model representation is annotated with an error message to give the user the chance to correct his error. Fig. 3.3 shows the editing process.

Modern text editors further enhance the user experience with not necessarily essential but yet important functionality. Some of this functionality is independent from the concrete notation, like *copy-paste*, *redo-undo*, etc. Other is notation dependant, like highlighting specific syntactical elements, like keywords, literals, or corresponding parenthesis. Also highlighting of semantic relations falls into this category: showing connections between identifier definitions and use, distinguishing identifier of different kind, etc. Beside enhancing the textual representations, editors might provide specific assistance in changing the representation. This is commonly known as *content assist* or *code completion*.

This is all editor functionality applied to editing model representation (thereby editing the model indirectly) and must not be confused with model

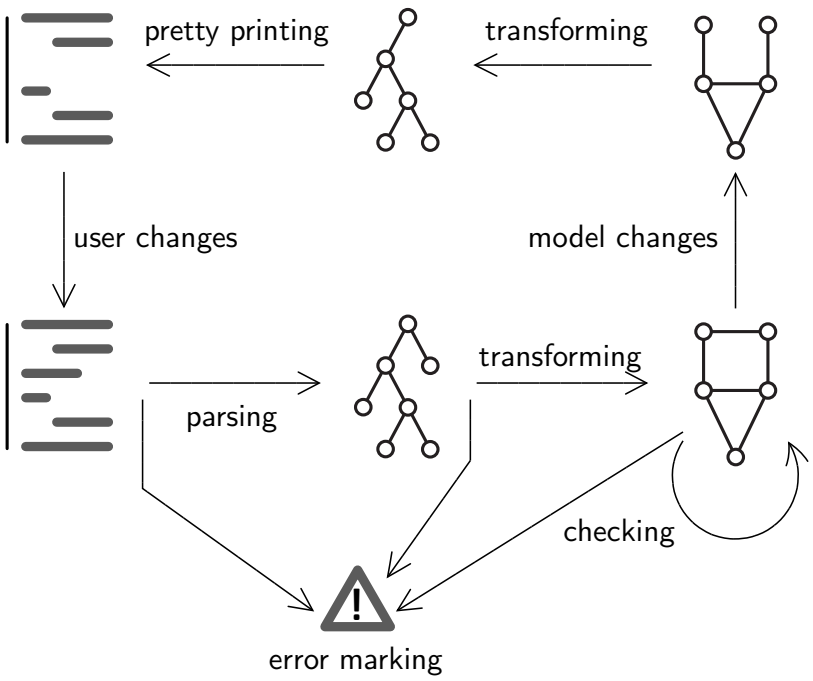


Figure 3.3: Editors combine parsing and pretty printing in a continuous process.

manipulation. *Refactoring* for example allows to change models in a controlled way, using predefined actions called *adaptations*. This and other operations performed on a model directly are different from representation changes. In this case the representation is only changed indirectly. Model editing based on a textual representation is often mixed with direct model editing capabilities.

3.3.3 Problems with Textual Notations

Textual Model Editors are Expensive

Describing a notation language and a corresponding mapping to a language description are the prerequisites to use a notation. But this is only a first step in realising effective programming or modelling based on such notations. Language developers have to create editing tools that bring as much assistance as possible to the language users. This assistance comprises syntax highlighting, outline views, annotation of syntactical and semantic errors, occurrence annotations, content assist, code formatting, and so on. Today, this level of assistance is only reached by modern programming environments

with highly capable language-dependent text editors. These modern program editors are complex and have to be built for each language independently. This renders the manual development of such editors too expensive for the textual parts of graphical languages, especially those of DSLs.

Meta-Models are More Powerful than Grammars

In this monograph, I study textual notations for languages and formalisms based on a MOF-like meta-modelling formalisms. As you will see in section 3.5, the context-free grammar meta-modelling formalism and MOF-like meta-modelling formalisms describe languages of different complexity. Without giving to much results in advance, I say that the elements of languages defined with a MOF-like formalism describe directed labeled graphs, and the elements of languages defined with grammars only describe labeled trees.

A semantic mapping of a textual notation has to close this gap in expressiveness. The diversity of possible differences between parse trees and model makes it hard to find general concepts. Such concepts could provide the necessary abstractions for a description language for notation semantics. Without such a language, notation semantics is hard to describe and its generic realisation is even harder. As a result, development of editors and parsers means extensive efforts to manually program language and notation dependant transformations between parse trees and models.

Textual Representations Are Often Partial Representations Embedded into a Graphical Representation

Graphical notations rarely consist entirely of symbols and connections, but usually contain pieces of structured text. These pieces can be described by a textual notation based on a context-free grammar formalism. A graphical notation therefore contains several textual sub-notations. This needs a formalism that allows to combine graphical and textual notations and frameworks that allow to create editors based on notations in this formalism.

High Quality Content Assist Depends on Language and Notation

Content assist provides the user of textual editors with snippets of possible follow up texts usually based on the context given by the cursor position. This text snippets are called *proposals*. These proposals can be put into two quality categories. Firstly, the proposal does not necessary lead to a

valid model representation.[2] Secondly, it is guaranteed that the proposal is allowed in the given context. The second, higher quality, category is obviously notation-dependent. This means an editor realising such functionality has to be written for a specific notation.

This causes a problem originated in the fact that meta-models are more powerful than context-free grammars. It is hard to find abstractions for content assist concepts and therefore it is hard to describe content assistance. As a result, currently existing frameworks for textual editors do only contain rudimentary support for content assist (usually from the first quality category) and editors featuring high quality content assist require extensive manual implementation of content assistance.

Problems with Non-Unique Notations

Many notations allow to have multiple representations for the same model. If you can find a second different representation for one representation that represents the same model, I call this representation *non-unique*; I call a notation with non-unique representations a *non-unique* notation. There are two major reasons for non-unique textual notations: (1) additional information like comments or white spaces and (2) different concrete presentation options[3] for the same constructs.

Non-unique notations cause problems, when a model is to be pretty printed, because possible reverse notations are ambiguous. When you pretty print a model with multiple possible representations, you have to choose one presentation. Assuming that the used reverse notation is consistent, you can always choose the minimal representation. But a minimal representation can not contain any reasonable comments, or white-spaces that go beyond a default text-layout derived from some sort of coding conventions. This also means, that you always select one default presentation option (the option used in the minimal representation) for all constructs with different possible presentations.

Another problem is caused by annotations. Given an arbitrary representation and the represented model and annotator has to find positions and ranges for model elements within the representation. The problem thereby is, that the used annotator uses a reverse notation to map model elements to

[2] The most common content assist of this category is the so called *hippo*-completion. Independent from the cursor position, hippo-completion simply offers all the words existing in the current representation.

[3] The different representation options in a non-unique notation are often called syntactic sugar. The different representations allow the language user to choose a representation that depending on the context might be more favourable for human comprehension.

parts of the representation. When the original notation is non-unique, the reverse notation is ambiguous: the annotator cannot find the original representation parts based on the model. A possible solution are additional information that determines which representation parts represent which model elements. Pieces of such information are called *traces*. Attached questions are: how to define and store traces in models, how to extend formalisms and frameworks with a trace mechanism?

Non-unique representations can cause rather technical than fundamental problems, because describing and realising their semantic mapping means describing and realising an additional mechanism to choose the right representation (non-uniqueness) and handle traces.

Problems with Ambiguous Notations

A single representation might denote several models. Ambiguous notations cause a problem: when you parse an ambiguous representation, you have to choose one model. As a solution, you have to insist on consistent notations. This way, you always choose the most abstract model as the represented model. When you examine concrete techniques to build parsers, you will see that this is not a practical problem, because the used techniques automatically chose the most trivial model, which is in most cases the most abstract model.

3.4 Related Work

Since each instance of a computer language needs representation in any case, research about notations for computer languages is as old as these languages. Actually, in the beginning, computer languages were even defined by their notations. The first high-level programming languages were defined using syntax theory, introduced by Chomsky [15], especially context-free grammars.

Grammars as descriptions for textual notations are used in different ways. Firstly, they are used to build parsers for syntactical analysis of textual representations as part of background parsing editors and compilers. To build compilers [1] for programming languages, context-free grammars were extended to deal with their limited complexity, which did not allow to deal with *static semantics* concerns, like name resolution, type-checking, etc. Important for the work in this thesis are attributed grammars by Knuth [55], which inspired the grammar to meta-model mappings in section 3.7.2. Today, syntax theory, context-free grammars, compiler construction techniques, and

attributed grammars[4] are facilitated for DSM of textually notated languages and their tools. Lämmel and Klint describe the development of languages based on grammar-ware in [54]. Example frameworks for DSM of grammar-ware-based languages are safari [13] and ASF+SDL [123].

Secondly, grammars can be used to describe *syntax directed programming*. In syntax directed programming environments and corresponding editors, language instances are created through distinct commands to create, delete, and manipulate single language construct instances. Textual representations only show the current language instance, but editing commands do not change the representation, but the language instance directly. An corresponding representation just reacts to changes in the language instance and shows these changes respectively. This editing strategy, opposed to background parsing, is also known as *model view controller* (MVC). MVC means that the representation (view) is just a view on the language instance (model), and commands (controller) are used to control model changes directly.

Early frameworks that use MVC are either based on meta-languages that use annotated context-free grammars or templates to describe language constructs and their representation simultaneously. The MVC-based writing of programs with textual programming languages was called *syntax directed programming. The Cornell Program Synthesizer* [116] is an example with a meta-language based on templates, the *Program System Generator* (PSG) [7] and the *IPSEN* project [52] use grammars with annotations. An environment that combines syntax directed programming with editors or viewers for other views on the programmed problem, like expression trees, data type diagrams, flow graphs, and the symbol table, is PECAN [89]. PSD and IPSEN also support editing with background parsing due to problems with specific language constructs. Mathematical terms for example, when created via commands, have to be syntezised from the root to the leafs, which is often counter intuitive compared with the left to right understanding of mathematical expressions. This problem and the lack of performance of 80's computer systems probably caused the end of the syntax directed programming idea and only parsers and compilers were generated from grammars. Today, Simonyi in [110] and Dimitriev in [25] use the MVC strategy to realise syntax directed programming on modern computer systems. Simonyi's intentional software is based on grammars and syntax trees, and Dimitriev's Meta Programming System (MPS) realises MVC for MOF-like meta-modelling. The MVC methods are intrinsically different to the background parsing techniques presented here.

In this chapter, I mostly look at textual notations and their relations

[4]The whole field is also known as *grammar-ware* [125].

	meta-modelling framework	editor kinds	pretty-printing	references	code-completion	separate meta-model
TEF	emf	text, model	yes	simple, custom	simple, custom	yes
xText	emf	text	no	custom	custom	no
TCS	km3	text	yes	symbol table	no	yes
MC	emf	text	yes	no	no	no
safari	-	text	-	custom	custom	-

Table 3.1: TEF in comparison with other textual editing frameworks based on framework features.

to meta-models. Since the early MOF recommendations, research about the relationship between grammar-ware and meta-model-based language descriptions[5] was conducted. The result were elaborations on the relationships between meta-models and grammars, and mappings between such language descriptions. This includes Alanen et al. [2], Scheidgen et al. [31], Wimmer et al. [125]. Due to a lack of formalisations for MOF-like meta-modelling, this work only comprises informal mappings between grammars and meta-models. This basic research was later utilised in frameworks for textual notations and corresponding tools: TCS [48, 49], TCSSL [69], MontiCore [59, 58, 60], xText [83], and Kleppe [53]. More details about this work and its relations to the material in this thesis can be found in 3.7.6.

Especially frameworks that allow DSM of feature rich, syntax and static semantics aware, editors for the eclipse platform are very popular. Fig. 3.1 compares the frameworks TEF [104], xText [83], TCS [49], MontiCore (MC) [60], and safari [45]. My framework is TEF (highlighted) and is based of research presented in this chapter (see 3.7). The table shows the used MOF-like meta-modelling framework (safari is only based on grammars and abstract syntax trees); whether the framework only allows to create text editors; supports pretty-printing of created models; whether it supports ref-

[5]Conclusively also known as *model-ware* [125].

erences, code-completion, and allows to create an editor for an existing meta-model. Frameworks typically either offer simple model wide references and code-completion (simple) or allow to manually program (custom) reference resolution and code-completion. TCS, however, uses a built-in symbol-table to resolve references. Frameworks that not allow to create editors for existing meta-models usually generate their own meta-models: The language developer has to describe notation and language with the same meta-language. These frameworks do not distinguish between notation and language description. The meta-model and corresponding models, created through background parsing, are only used to check and code-complete references, populate an outline-view, or format a textual representation with pretty-printing. Besides offering or not offering such features, none of the existing frameworks besides TEF allow to create model editors. Model editors are a prerequisite for embedding textual model editors into other editors, like graphical editors (see 3.7.5).

The second kind of representations, besides programming languages and their textual representations, are graphical representations, mainly used in process modelling and computer system modelling. Analogous to syntax theory, graph grammars were used to formalise graphical notations. Graph grammars with similar characteristics but different mathematical background exist: Ehrig gives an introduction to graph grammars based on algebraic specification in [27]. Besides using formal theory, graphical notations are often described informally. This is evident in the description of many popular languages, like UML [73, 74] and SDL [46]. Whereas, grammar-ware uses textual notations as language descriptions, graph grammars or even informal graphical notations could not prevail as language descriptions. This is one of the main motivation for the development of the OMG MOF standard (MOF-like meta-modelling), and the overall separation between notation and language description. Graph-based notation, languages, and corresponding tools can be developed with frameworks that are either based on graph grammars (Atom [22], Ehrig [29], Fujaba [71]) or annotated MOF-like meta-models (GME [21], GMF [119], GEF [118], XMF [12], metaEdit [121], KOGGE [26], and Microsoft Domain-Specific Language (DSL) Tools [64]).

3.5 The General Relationships Between Context-Free Grammars and Meta-Models

When you describe notations with grammars, and use these notations to represent language instances described with meta-models, you need to un-

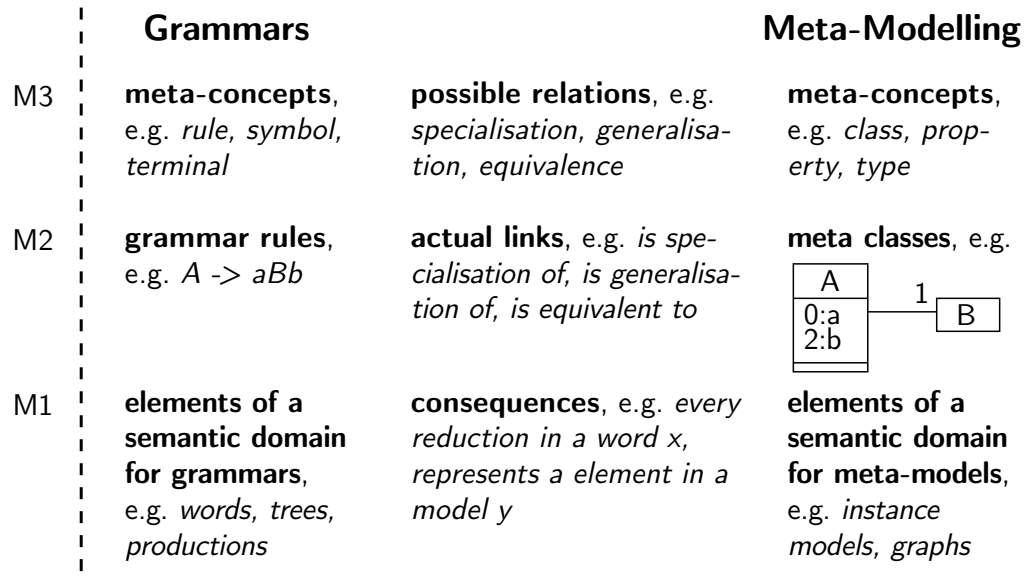


Figure 3.4: Informal display of the relations between the two modelling platforms and their meaning on the different modelling layers.

derstand how construct definitions in grammars and construct definitions in meta-models can be related to each other. This understanding will help to create meta-languages that allow to create meta-models from grammars, or to describe a mapping between a given grammar and a given meta-model. It is also a prerequisite to build meta-tools that will automatically transform textual language instances based on a grammar into instances of a meta-model and vice versa.

You can understand grammars as notation descriptions for models described with meta-models (refer to 3.2). On the other hand, you can see grammars and meta-models as language descriptions from two different meta-modelling formalisms (refer to 2.3.2). From this meta-modelling formalism perspective, you are looking at two formalisms that use two different meta-meta-models with different semantics. Hence, both formalisms use a different set of meta-modelling constructs to define means for describing language instances, and conclusively, both formalisms also use different semantic domains that form the set of describable languages (see Fig. 3.4).

Having two meta-modelling formalisms means you have two meta-modelling languages that follow a similar purpose but use different description means and semantics. Looking into the relationship between grammars and meta-models means to relate the description and semantic concepts of both formalisms. The semantic domains of both formalisms are founded on different mathematical grounds. Grammars use symbols and rules to describe

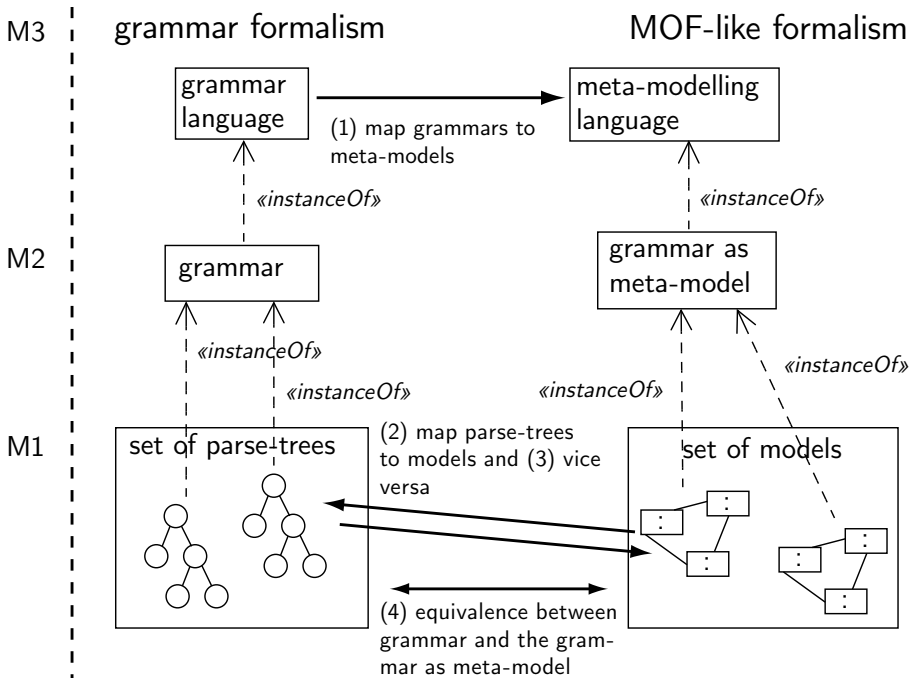


Figure 3.5: Different steps to show the relationships between grammars and meta-models.

languages; the semantics of grammars can be defined with rewrite systems, which lead to words, terms, productions, or parse trees [44]. Meta-models use classes and properties to describe languages, their semantics is based on an object-orientated data structure (see section 2.2.1), whose elements form graphs. Since both formalisms and respective platforms are based on different mathematical concepts, and result in disjoint meta-modelling technologies: both formalisms define two different *technological spaces*.

To bridge both technological spaces, you need to put them on the same semantic (mathematical) grounds. As a result, you can transcribe an instance of a grammar into an instance of a meta-model (M1). The possibility to identify instances of a grammar with instances of a meta-model allows to define equivalence between grammars and meta-models (M2), which finally allows to argue about the relations between grammar and meta-modelling constructs (M3).

I assume the following hypothesis: for each grammar you can find a meta-model so that for each instance produced by the grammar exists an corresponding meta-model instance. That means, I assume that meta-models are at least as expressive as grammars.

In the following sub-sections, I do the following to prove this hypothesis:

I formally define a formalism for grammars, as I did with MOF-like meta-modelling in section 2.3.2. Based on that, I (1) define a mapping from MOF-like meta-models to grammars; (2, 3) define morphisms between grammar instances and meta-model instances; (4) show that these morphisms form a homomorphism between the set of all grammar instances and all grammar-derived meta-model instances. Fig. 3.5 show these steps and the involved entities on the different meta-layers.

3.5.1 A Mapping from Grammars to MOF-like Meta-Models

In this rather mathematical section, I lay the needed foundation for creating equivalent meta-models from existing grammars as described earlier.

Formal Model for Grammars

Definition 30 (grammar, *math.*) *A grammar $\mathfrak{G}$ is a tuple:*

$$\mathfrak{G} = (Symbols, NonTerminals, Rules, lhs, rhs)$$

with

$$\begin{aligned} NonTerminals &\subseteq Symbols \\ lhs &: Rules \to NonTerminals \\ rhs &: Rules \times \mathbb{N} \to Symbols \cup \{\star\} \end{aligned}$$

For grammar $\mathfrak{G}$ to be a valid context-free grammar certain constraints must hold. For all $r \in Rules$, $i \in \mathbb{N}$ the following formulas must evaluate to *true*:

$$rhs(r, i) = \star \Rightarrow rhs(r, i + 1) = \star \tag{3.1}$$

$$rhs(r, 0) \neq \star \tag{3.2}$$

The first formula (3.1) ensures that the right hand side of a rule is a gapless sequence of symbols. The second formula (3.2) states that each right hand side of a rule must at least comprise one symbol.

Parse Trees

Based on a context-free grammar $\mathfrak{G}$, I can define what a parse tree is.

Definition 31 (parse tree, *math.*) *A parse tree is a tuple*

$$T = (Grammar, Nodes, Edges, symbol, rule)$$

with:

$$\begin{array}{rcl} Grammar & & \text{is a grammar} \\ Edges & \subseteq & Nodes \times Nodes \times \mathbb{N} \\ symbol & : & Nodes \to Symbols_{Grammar} \\ rule & : & Nodes \to Rules_{Grammar} \cup \{\star\} \end{array}$$

A tuple $edge \in Edges$ *has the form*

$$edge = (parent, child, index)$$

Each parse tree T can denote an actual instance of the corresponding grammar. A parse tree T is an instance of grammar $\mathfrak{G}$ if $Grammar_T = \mathfrak{G}$. Each inner node (a node with children) instantiates a rule. Each leave node (a node without children) represents a terminal. A node that instantiates a rule also represents the non-terminal symbol on the left side of the instantiated rule. A parse tree does not have to start with a designated *start* symbol and can contain multiple trees, i.e. multiple root nodes. Intuitively, this means that the parse trees do not only contain a tree with a designated *start* symbol for its root node, but also all sub-trees of that tree. This is different from the usual understanding of the semantics of context-free grammars and is done to make context-free grammars and meta-models more comparable. You could also alter the normal semantics of meta-models: you have to constrain models (instances of meta-models) in such a way, that they always must contain a value based on a designated *start* class, and that values and links must form a connected graph. In the later, I assume that neither, the modified context-free grammar semantics, nor a possible change in the meta-modelling semantics, change the expressiveness of both formalisms significantly.

Of course, the grammars constrains the ways in which edges can connect nodes. Before I come to these constraints, I define the following left-total helper relations:

$$\begin{array}{rcl} children & \subseteq & Nodes \times Nodes \\ children(n) & =_{def} & \{n' | \exists e \in Edges, i \in \mathbb{N} : e = (n, n', i)\} \end{array} \tag{3.3}$$

$$
\begin{aligned}
allChildren &\subseteq Nodes \times Nodes && (3.4)\\
allChildren &=_{def} \{n' | n' \in children(n) \vee \\
&\qquad \exists n'' \in children(n) : n' \in allChildren(n'')\} \\
root &: Nodes \to Nodes && (3.5)\\
root(n) &=_{def} \begin{cases} n & \text{if } \nexists n', e, i : (n', n, i) = e \\ root(n') & \text{else with } \exists e, i : (n', n, i) = e \end{cases}
\end{aligned}
$$

With *children* (3.3), you can access all nodes that are children of one node. With *allChildren* (3.4), you can access all child nodes of a node, including children's children, and so on. The function *root* (3.5) gives you the root node of a tree.

For each parse tree T to be a valid parse tree of grammar $\mathfrak{G} = Grammar_T$, certain constraints must hold. For all $e, e_1, e_2 \in Edges$, $n, n_1, n_2 \in Nodes$, $r \in Rules_{Grammar}$, and $i, j \in \mathbb{N}$ the following formulas must evaluate to *true*:

$$
\begin{aligned}
& n \notin allChildren(n) && (3.6)\\
& e_1 = (n_1, n, i) \wedge e_2 = (n_2, n, j) \Rightarrow e_1 = e_2 && (3.7)\\
& e_1 = (n, n_1, i) \wedge e_2 = (n, n_2, j) \wedge e_1 \neq e_2 \Rightarrow i \neq j && (3.8)\\
& rule(n) = \star \Rightarrow symbol(n) \in Symbols \setminus NonTerminals && (3.9)\\
& rule(n) \neq \star \Rightarrow symbol(n) = lhs_{Grammar}(rule(n)) && (3.10)\\
& rule(n) = r \neq \star \Rightarrow rhs_{Grammar}(r, i) \neq \star \Rightarrow && (3.11)\\
&\qquad \exists e, n' : (n, n', i) = e \wedge symbol(n') = rhs(r, i)
\end{aligned}
$$

The first two constraints 3.6 and 3.7 guarantee that a parse tree does not contain circles and each node has only one parent, i.e. the parse tree is indeed a tree. The third constraint 3.8 makes sure that among a nodes outgoing edges an index is only used once. Constraints 3.9 and 3.10 say that an inner node represents the left-hand symbol of its rule and an outer node represents a terminal symbol. The last constraint 3.11 is to verify that each node actually instantiates a rule and child nodes instantiate rules with corresponding left hand symbols. Since $lhs_G(r, 0) \neq \star$ for all grammars G and rules r, a node with rule is always an inner node based on the last constraint.

A Formalism for Context-Free Grammars

Definition 32 (context-free grammar formalism, *math.*)
A formalism for context-free grammars $CFG = (L, SM, SD)$, is define by the three components language L, semantic mapping SM, and semantic domain SD. The language is the set of all grammars:

$$L_{CFG} =_{def} \{\mathfrak{G} | \mathfrak{G} \text{ is a context-free grammar}\}$$

The semantic domain is the set of all sets of parse trees for all context-free grammars:

$$SD_{CFG} =_{def} \bigcup_{\mathfrak{G} \in L} \{T | Grammar_T = \mathfrak{G}\}$$

Given a grammar $\mathfrak{G}$, I define the set of all instances of this grammar $trees(\mathfrak{G})$, i.e. the set of all parse trees for this grammar as

$$trees(\mathfrak{G}) =_{def} \{T | T \text{ is a parse tree } \wedge Grammar_T = \mathfrak{G}\}$$

This function trees defines a mapping between L and SD:

$$SM_{CFG} =_{def} trees$$

A Mapping from Context-Free Grammars to Meta-Models

In the last sections, I completed the first step (see figure 3.5): I defined a formal model for grammars. As the next step, I map grammars on meta-models, i.e. define a meta-model $mapping(\mathfrak{G})$ for each context-free grammar $\mathfrak{G}$. A few remarks on this mapping:

- Symbols denote types: using parse trees as semantic domain for grammars, non-terminals define types for the inner nodes of such a tree, terminals define the types of leafs. Non-terminals are abstract types; the concrete forms of non-terminals are rules (an inner parse tree node and its children define instances of rules). Terminals define concrete primitive types.

- A rule is a concrete class with no further sub-types and it is always a sub-class of the non-terminal on its left-hand side. The instances of a rule describe inner nodes in a parse tree.

- The different items on a rule's right-hand side are properties of the rule class. The rule's class' attribute values denote the children of the corresponding node. For each symbol on the rule's right-hand side the

node has to have exactly one child. You can also say that each child is owned by the node, because parse trees are acyclic. Therefore each right-hand side property has a multiplicity of exactly one and always has composite semantics.

Before I actually define the mapping, I need further helpers. Firstly, I assume that for each $\mathfrak{G}$, you can find a total reversible function $unique_{\mathfrak{G}} : Rules_{\mathfrak{G}} \cup Symbols_{\mathfrak{G}} \to \mathbb{S}$. Secondly, I define $str : \mathbb{N} \to \mathbb{S}$ to be a total reversible function. Thirdly, I define the following short-hand notations

$$
\begin{aligned}
symbolClass_{\mathfrak{G}}(s) &= (unique_{\mathfrak{G}}(s), true) \\
ruleClass_{\mathfrak{G}}(r) &= (unique_{\mathfrak{G}}(r), false) \\
type_{\mathfrak{G}}(a) &= \begin{cases} ruleClass_{\mathfrak{G}}(a) & \text{if } a \in Rules_{\mathfrak{G}} \\ symbolClass_{\mathfrak{G}}(a) & \text{if } a \in NonTerminals_{\mathfrak{G}} \\ a & \text{if } a \in Symbols_{\mathfrak{G}} \setminus NonTerminals_{\mathfrak{G}} \end{cases}
\end{aligned}
$$

I define for each grammar $\mathfrak{G}$ a MOF-like meta-model. Given a grammar G, I define $mapping(\mathfrak{G}) = \mathfrak{M}$ as follows:

$$
\begin{aligned}
Classes_{\mathfrak{M}} &=_{def} \{symbolClass_{\mathfrak{G}}(s) | s \in NonTerminals_{\mathfrak{G}}\} \cup \\
&\quad \{ruleClass_{\mathfrak{G}}(r) | r \in Rules_{\mathfrak{G}}\} \\
Types_{\mathfrak{M}} &=_{def} Classes_{\mathfrak{M}} \cup Symbols_{\mathfrak{G}} \setminus NonTerminals_{\mathfrak{G}} \\
Props_{\mathfrak{M}} &=_{def} \{(str(i), ruleClass_{\mathfrak{G}}(r), t, true, false, 1, 1) | \\
&\quad\quad r \in Rules_{\mathfrak{G}} \wedge t = type_{\mathfrak{G}}(rhs_{\mathfrak{G}}(r, i))\} \\
superTypes_{\mathfrak{M}} &=_{def} \{(symbolClass_{\mathfrak{G}}(s), ruleClass_{\mathfrak{G}}(r)) | lhs_{\mathfrak{G}} = s\}
\end{aligned}
$$

Formally, I still need to show that the mapping result really is a valid meta-model.

Theorem 3.1 *If G is a valid context-free grammar $\mathfrak{M} = mapping(G)$ is a valid meta-model.*

Proof 1 *I have to show that all meta-model constraints 2.4 to 2.7 evaluate to true for $\mathfrak{M}$ if $\mathfrak{G}$ is valid. s*

property names, 2.4:

$$
\begin{aligned}
& p_i \in allProps_{\mathfrak{M}}(c) \Rightarrow owner_{p_i} \in allSuperTypes(c) \cup \{c\} \\
\Rightarrow\ & \exists r : owner_{p_i} = ruleClass_{\mathfrak{G}}(r) \vee owner_{p_i} \\
& \qquad\qquad = symbolClass_{\mathfrak{G}}(lhs_{\mathfrak{G}}(r)) \\
\Rightarrow\ & \exists r : owner_{p_i} = ruleClass_{\mathfrak{G}}(r)
\end{aligned}
$$

$$\begin{aligned}\Rightarrow \quad name_{p_1} = name_{p_2} &\Rightarrow \exists i : str(i) = name_{p_1} = name_{p_2} \\ &\Rightarrow p_1 = p_2 \\ &\quad = (str(i), r, rhs_{\mathfrak{G}}(r,i), true, false, 1, 1)\end{aligned}$$

class names, 2.5:

$$\begin{aligned}name_{c_1} = name_{c_2} \quad &\Rightarrow \quad \exists a : unique_{\mathfrak{G}}(a) = name_{c_1} = name_{c_2} \wedge \\ &\qquad a \in Symbols_{\mathfrak{G}} \\ &\Rightarrow \quad c_1 = c_2 = (unique_{\mathfrak{G}}(a), true) \wedge a \in Rules_{\mathfrak{G}} \\ &\Rightarrow \quad c_1 = c_2 = (unique_{\mathfrak{G}}(a), false)\end{aligned}$$

acyclic super types, 2.6:

$$\begin{aligned}isAbstract_c \quad &\Rightarrow \quad \exists s : c = symbolClass_{\mathfrak{G}}(s) \\ &\Rightarrow \quad c \notin allSuperTypes(c) = \emptyset \\ \neg isAbstract_c \quad &\Rightarrow \quad \exists r : c = ruleClass_{\mathfrak{G}}(r) \wedge \\ &\qquad (allSuperTypes(c) = \{symbolClass_{\mathfrak{G}}(lhs_{\mathfrak{G}}(r))\} \\ &\qquad\quad \Rightarrow c \notin allSuperTypes(c))\end{aligned}$$

reasonable bounds, 2.7:

$$lower_p = upper_p = 1 \Rightarrow lower_p \leq upper_p$$

3.5.2 Semantic Equivalence of Grammar and Meta-Model

To show that a Grammar $\mathfrak{G}$ and corresponding meta-model $mapping(\mathfrak{G})$ are equivalent, I show that $trees(\mathfrak{G})$ and $models(mapping(\mathfrak{G}))$ are equivalent, i.e. homomorph. I define a morphism $h : trees(\mathfrak{G}) \rightarrow models(mapping(\mathfrak{G}))$ and $h^{-1} : models(mapping(\mathfrak{G})) \rightarrow trees(\mathfrak{G})$ and show that for all $T \in trees(\mathfrak{G})$ $h^{-1}(h(T)) = T$, and for all $M \in models(mapping(\mathfrak{G}))$ $h(h^{-1}(M)) = M$. With morphism in each directions grammar and meta-model are homomorph, hence equivalent.

Mapping Parse Trees on Models

I define $h : trees(\mathfrak{G}) \rightarrow models(mapping(\mathfrak{G}))$ as follows:

$$\begin{aligned}MetaModel_{h(T)} &=_{def} mapping(\mathfrak{G}) \\ Values_{h(T)} &=_{def} Nodes_T \\ Links_{h(T)} &=_{def} \{(n, n', 0) | (n, n', i) \in Edges_T\}\end{aligned}$$

$$
\begin{aligned}
metaType_{h(T)}(n) &=_{def} \begin{cases} ruleClass_{\mathfrak{G}}(rule_T(n)) & \text{if } rule_T(n) \neq \star \\ symbol_T(n) & \text{else} \end{cases} \\
metaProp_{h(T)} &=_{def} \{((n, n', 0), (str(i), r, t, true, false, 1, 1))| \\
&\qquad e = (n, n', i) \in Edges_T \wedge \\
&\qquad r = ruleClass_{\mathfrak{G}}(rule_T(n)) \wedge \\
&\qquad t = type_{\mathfrak{G}}(symbol_T(n'))\}
\end{aligned}
$$

I need to show that the produced models are actually models.

Theorem 3.2 *Given grammar $\mathfrak{G}$ and meta-model $\mathfrak{M} = mapping(\mathfrak{G})$, for all $T \in trees(\mathfrak{G})$ $h(T)$ is a valid model.*

Proof 2 *I have to show that all model constraints 2.10 to 2.16 evaluate to true for $h(T)$. s*

only object nodes have links, 2.10:

$$
\begin{aligned}
& \exists e \in Edges_T \Rightarrow rule_T(src_l) \neq \star \\
\Rightarrow\ & metaType_{h(T)}(src_l) = ruleClass_{\mathfrak{G}}(rule_T(src_l)) \\
& \{ruleClass_{\mathfrak{G}}(r)|r \in Rules_{\mathfrak{G}}\} \subseteq Classes_{\mathfrak{M}} \\
\Rightarrow\ & metaType_{h(T)}(src_l) \in Classes_{\mathfrak{M}}
\end{aligned}
$$

no abstract instantiation, 2.11:

$$
\begin{aligned}
& metaType_{h(T)} \in Classes_{\mathfrak{M}} \wedge Classes_{\mathfrak{M}} \cap Symbols_{\mathfrak{G}} = \emptyset \\
\Rightarrow\ & metaType_{h(T)}(v) = ruleClass_{\mathfrak{G}}(rule_T(v)) \\
\Rightarrow\ & isAbstract_{metaType(v)} = false
\end{aligned}
$$

property types, 2.12:

$$
\begin{aligned}
& c = metaType_{h(T)}(src_l) \\
\Rightarrow\ & c = ruleClass_{\mathfrak{G}}(rule_T(src_l)) \wedge l = (src_l, trg_l, 0) \\
\Rightarrow\ & metaProp_{h(T)}(l) = (str(i), c, type_{\mathfrak{G}}(symbol_T(src_l)), \\
& \qquad\qquad true, false, 1, 1) \\
\Rightarrow\ & metaProp_{h(T)}(l) \in allProps_{\mathfrak{M}}(c) \\
\Rightarrow\ & type_{metaProp_{h(T)}(l)} = type_{\mathfrak{G}}(symbol_T(src_l)) \\
if\ & metaType_{h(T)}(trg_l) = ruleClass_{\mathfrak{G}}(rule_T(trg_l))
\end{aligned}
$$

$$
\begin{aligned}
&\Rightarrow\ type_{\mathfrak{G}}(symbol_T(trg_l)) \in \\
&\qquad\qquad superTypes_{\mathfrak{M}}(ruleClass_{\mathfrak{G}}(rule_T(trg_l))) \\
&\Rightarrow\ metaType_{h(T)}(trg_l) \in assignables(type_{metaProp_{h(T)}(l)}) \\
\textit{else if}\quad & metaType_{h(T)}(trg_l) = symbol_T(trg_l) \\
&\Rightarrow\ metaType_{h(T)}(trg_l) \in assignables(type_{metaProp_{h(T)}(l)})
\end{aligned}
$$

ordering, 2.13:

$$
\begin{aligned}
&\forall p \in Props_{\mathfrak{M}} : isOrdered_p = false \wedge \forall l \in Links_{h(T)} : index_l = 0 \\
\Rightarrow\ &\forall l_1, l_2 : \neg isOrdered_p \wedge index_{l_1} = index_{l_2} = 0
\end{aligned}
$$

multiplicity, 2.14:

$$
\begin{aligned}
&e_1 = (src_l, n_1, i) \wedge e_2 = (src_l, n_2, i) \wedge i \neq j \\
\Rightarrow\ &metaProp_{h(T)}((src_l, n_1, 0)) \neq metaProp_{h(T)}((src_l, n_2, 0)) \\
\Rightarrow\ &\forall v, p : |\{l | src_l = v \wedge metaProp_{h(T)}(l) = p\}| = 1 \\
&\forall p \in Props_{\mathfrak{M}} : lower_p = upper_p = 1 \\
\Rightarrow\ &1 = lower_p \leq |\{l | src_l = v \wedge metaProp_{h(T)}(l) = p\}| \leq upper_p = 1
\end{aligned}
$$

composition, 2.15:

$$
\begin{aligned}
&n \notin allChildren(n) \wedge \forall p \in Props_{\mathfrak{M}} : isComposite_p \\
\Rightarrow\ &n \notin allComponents(n)
\end{aligned}
$$

only one container, 2.16:

$$
\begin{aligned}
&e_1 = (n_1, n, i) \wedge e_2 = (n_2, n, i) \Rightarrow e_1 = e_2 \\
\Rightarrow\ &l_1 = (n_1, n, 0) \wedge l_2 = (n_2, n, 0) \Rightarrow l_1 = l_2 \Rightarrow n_1 = n_2
\end{aligned}
$$

Mapping Models to Parse Trees

I define $h^{-1} : models(mapping(\mathfrak{G})) \rightarrow trees(\mathfrak{G})$ as follows:

$$
\begin{aligned}
Grammar_{h^{-1}(M)} &=_{def} \mathfrak{G} \\
Nodes_{h^{-1}(M)} &=_{def} Values_M \\
Edges_{h^{-1}(M)} &=_{def} \{(n, n', i) | (n, n', 0) \in Links_M \wedge \\
&\qquad\qquad name_{metaProp_M(n,n',0)} = str(i)\}
\end{aligned}
$$

$$rule_{h^{-1}(M)}(n) =_{def} \begin{cases} r & \text{if } metaType_M(n) = (unique_{\mathfrak{G}}(r), false) \\ \star & \text{else} \end{cases}$$

$$symbol_{h^{-1}(M)}(n) =_{def} \begin{cases} lhs(r) & \text{if } metaType_M(n) = \\ & \quad (unique_{\mathfrak{G}}(r), false) \\ metaType_M(n) & \text{else} \end{cases}$$

I need to show that the produced parse trees are actually parse trees.

Theorem 3.3 *For all $M \in models(mapping(\mathfrak{G}))$ and $T = h^{-1}(M)$ T is a valid parse tree.*

Proof 3 *I have to show that all parse tree constraints 3.6 to 3.11 evaluate to true for $h^{-1}(M)$. s*

circle free, 3.6:

$$\begin{aligned} & n \notin allComponents(n) \wedge \forall p \in Props_{\mathfrak{M}} : isComposite_p \\ \Rightarrow \quad & n \notin allChildren(n) \end{aligned}$$

one parent, 3.7:

$$\begin{aligned} & \forall p \in Props_{\mathfrak{M}} : isComposite_p \\ \Rightarrow \quad & l_1 = (n_1, n, 0) \wedge l_2 = (n_2, n, 0) \Rightarrow n_1 = n_2 \\ \Rightarrow \quad & e_1 = (n_1, n, i) \wedge e_2 = (n_1, n, j) \Rightarrow n_1 = n_2 \\ \Rightarrow \quad & metaProp_M((n_1, n, 0)) = (str(i), \dots) = \\ & \qquad metaProp_M((n_2, n, 0)) = (str(j), \dots) \\ \Rightarrow \quad & i = j \Rightarrow e_1 = e_2 \end{aligned}$$

single index, 3.8:

$$\begin{aligned} & e_1 = (n, n_1, i) \wedge e_2 = (n, n_2, j) \wedge e_1 \neq e_2 \wedge i = j \\ \Rightarrow \quad & n_1 \neq n_2 \\ & \forall p \in Props_{\mathfrak{M}} : upper_p = 1 \\ \Rightarrow \quad & metaProp_M((n, n_1, 0)) \neq metaProp_M((n, n_2, 0)) \\ \Rightarrow \quad & name_{metaProp_M((n,n_1,0))} = str(i) \neq str(j) \\ & \qquad\qquad \neq name_{metaProp_M((n,n_2,0))} \\ \Rightarrow \quad & i \neq j \end{aligned}$$

leafs are terminals, 3.9:

$$\begin{aligned}
& rule_{h^{-1}(M)}(n) = \star \\
\Rightarrow\ & symbol_{h^{-1}(M)}(n) = metaType_M(n) \\
& \qquad \wedge \nexists r : metaType_M(n) = ruleClass_{\mathfrak{G}}(r) \\
& \forall s : isAbstract_{symbolClass_{\mathfrak{G}}(s)} \\
\Rightarrow\ & metaType_M(n) \notin Classes_{\mathfrak{M}} \\
\Rightarrow\ & metaType_M(n) \in Symbols_{\mathfrak{G}} \setminus NonTerminals_{\mathfrak{G}} \\
\Rightarrow\ & symbol_{h^{-1}(M)}(n) \in Symbols_{\mathfrak{G}} \setminus NonTerminals_{\mathfrak{G}}
\end{aligned}$$

inner nodes are non-terminals, 3.10:

$$\begin{aligned}
& rule_{h^{-1}(M)}(n) \neq \star \\
\Rightarrow\ & symbol_{h^{-1}(n)} = lhs_{\mathfrak{G}}(r) \wedge \\
& \qquad metaType_M(n) = ruleClass_{\mathfrak{G}}(r) \\
\Rightarrow\ & rule_{h^{-1}(M)}(n) = r \wedge metaType_M(n) = ruleClass_{\mathfrak{G}}(r) \\
\Rightarrow\ & symbol_{h^{-1}(M)}(n) = lhs_{\mathfrak{G}}(rule_{h^{-1}(M)}(n))
\end{aligned}$$

rule instantiation, 3.11:

$$\begin{aligned}
& rule_{h^{-1}(M)}(n) = r \neq \star \\
\Rightarrow\ & metaType_M(n) = ruleClass_{\mathfrak{G}}(r) \\
& rhs_{\mathfrak{G}}(r,i) \neq \star \\
\Rightarrow\ & \exists p, n' : p = (str(i), ruleClass_{\mathfrak{G}}(r), type_{\mathfrak{G}}(rhs_{\mathfrak{G}}(r,i)), \\
& \qquad\qquad true, false, 1, 1) \\
\Rightarrow\ & \exists l : l = (n, n', 0) \wedge metaType_M(n') = type_{\mathfrak{G}}(rhs_{\mathfrak{G}}(r,i)) \\
\textit{if}\quad & metaType_M(n') \notin Classes_{\mathfrak{M}} \\
\Rightarrow\ & \exists e, i : e = (n, n', i) \wedge symbol_{h^{-1}}(n') = rhs_{\mathfrak{G}}(r,i) \\
\textit{else}\quad & \exists r' : metaType_M(n') = ruleClass_{\mathfrak{G}}(r') \\
\Rightarrow\ & \exists e, i : e = (n, n', i) \wedge symbol_{h^{-1}(M)} = lhs_{\mathfrak{G}}(r') = rhs_{\mathfrak{G}}(r,i)
\end{aligned}$$

Equivalence

I finally show that h^{-1} is the inverse of h and that, therefore, $trees(\mathfrak{G})$ and $model(mapping(\mathfrak{G}))$ are homomorph.

Theorem 3.4 *$h^{-1}(h(T)) = T$ and $h(h^{-1}(M)) = (M)$ for all $T \in trees(\mathfrak{G})$ and $M \in models(mapping(\mathfrak{G}))$.*

Proof 4 *Based on the definition of h and h^{-1} I need to show that:*

$$\begin{aligned} Edges_{h^{-1}(h(T))} &= Edges_T \\ rule_{h^{-1}(h(T))}(n) &= rule_t(n) \\ symbol_{h^{-1}(h(T))}(n) &= symbol_t(n) \end{aligned}$$

and

$$\begin{aligned} Links_{h^(h^{-1}(M))} &= Links_M \\ metaType_{h(h^{-1}(M))}(n) &= metaType_M(n) \\ metaProp_{h(h^{-1}(M))}(l) &= metaProp_M(l) \end{aligned}$$

s

$Edges_{h^{-1}(h(T))} = Edges_T$*:*

$$\begin{aligned} & Edges_{h^{-1}(h(T))} \\ = \; & \{(n,n',i)|(n,n',0) \in Links_{h(T)} \wedge \\ & \qquad name_{metaProp_{h(T)}((n,n',0))} = str(i)\} \\ & Links_{h(T)} = \{(n,n',0)|(n,n',i) \in Edges_T\} \\ \Rightarrow \; & Edges_{h^{-1}(h(T))} = Edges_T \\ \Leftrightarrow \; & \forall (n,n',i) \in Edges_T : name_{metaProp_{h(T)}((n,n',0))} = str(i) \\ & definition\ of\ metaProp_{h(T)} \\ \Rightarrow \; & (n,n',i) \in Edges_T \Rightarrow name_{metaProp_{h(T)}((n,n',0))} = str(i) \\ \Rightarrow \; & Edges_{h^{-1}(h(T))} = Edges_T \end{aligned}$$

$rule_{h^{-1}(h(T))}(n) = rule_T(n)$*:*

$$\begin{aligned} & & & definition\ of\ rule_{h^{-1}(h(T))}\ and\ metaType_{h(T)} \Rightarrow \\ & if & & rule_T(n) = \star \\ & & \Rightarrow \; & metaType_{h(T)}(n) = symbol_T(n) \\ & & \Rightarrow \; & metaType_{h(T)}(n) \neq (unique_{\mathfrak{G}}(r), false) \\ & & \Rightarrow \; & rule_{h^{-1}(h(T))} = \star \Leftrightarrow rule_T(n) = \star \\ & else & & \\ & & \Rightarrow \; & metaType_{h(T)}(n) = ruleClass_{\mathfrak{G}}(rule_T(n)) \\ & & \Rightarrow \; & rule_{h^{-1}(h(T))}(n) = rule_T(n) \end{aligned}$$

$symbol_{h^{-1}(h(T))}(n) = symbol_T(n)$:

$$
\begin{array}{lll}
 & & \textit{definition of } symbol_{h^{-1}(h(T))} \textit{ and } metaType_{h(T)} \Rightarrow \\
 & \textit{if} & rule_T(n) \neq \star \\
 & \Rightarrow & metaType_{h(T)}(n) = ruleClass_{\mathfrak{G}}(rule_T(n)) \\
 & \Rightarrow & metaType_{h(T)}(n) = (unique_{\mathfrak{G}}(rule_T(n)), false) \\
 & \Rightarrow & symbol_{h^{-1}(h(T))}(n) = lhs_{\mathfrak{G}}(rule_T(n)) \\
 & \Rightarrow & symbol_{h^{-1}(h(T))}(n) = symbol_T(n) \\
\textit{else} & & \\
 & \Rightarrow & metaType_{h(T)}(n) = symbol_T(n) \\
 & \Rightarrow & symbol_{h^{-1}(h(T))}(n) = symbol_T(n)
\end{array}
$$

$Links_{h(h^{-1}(M))} = Links_M$:

$$
\begin{array}{lll}
 & & Links_{h(h^{-1}(M))} \\
 & = & \{(n, n', 0) | (n, n', i) \in Edges_{h^{-1}(M)}\} \\
 & & Edges_{h^{-1}(M)} = \{(n, n', i) | (n, n', i) \in Links_M \wedge \\
 & & \qquad\qquad name_{metaProp_M((n,n',0))} = str(i)\} \\
 & \Rightarrow & Links_{h(h^{-1}(M))} = Links_M \\
 & \Leftrightarrow & \forall (n, n', 0) \in Links_M : \exists i : name_{metaProp_M((n,n',0))} = str(i) \\
 & & metaProp \textit{ is total and } name_{metaProp(\ldots)} \in \{str(i) | i \in \mathbb{N}\} \\
 & \Rightarrow & Links_{h(h^{-1}(M))} = Links_M
\end{array}
$$

$metaType_{h(h^{-1}(M))}(n) = metaType_M(n)$:

$$
\begin{array}{lll}
 & & \textit{definition of } metaType_{h(h^{-1}(M))} \textit{ and } rule_{h^{-1}(M)} \\
 & \textit{if} & metaType_M(n) = (unique_{\mathfrak{G}}(r), false) \\
 & \Rightarrow & metaType_{h(h^{-1}(M))}(n) = ruleClass_{\mathfrak{G}}(r) \\
 & & \qquad\qquad = (unique_{\mathfrak{G}}(r), false) = metaType_M(n) \\
\textit{else} & & metaType_{h(h^{-1}(M))}(n) = symbol_{h^{-1}(M)}(n) \\
 & \Rightarrow & metaType_{h(h^{-1})(M))}(n) = metaType_M(n)
\end{array}
$$

$metaProp_{h(h^{-1}(M))}(l) = metaProp_M(l)$:

$$
\begin{array}{l}
\textit{Property names are unique within a class:} \\
\forall l_1, l_2 \in dom(metaProp) :
\end{array}
$$

$$
\begin{aligned}
& name_{metaProp(l_2)} = name_{metaProp(l_2)} \wedge \\
& owner_{metaProp(l_1)} = owner_{metaProp(l_2)} \\
\Rightarrow\ & metaProp(l_1) = metaProp(l_2)
\end{aligned}
$$

This means I only need to show that

(1) $name_{metaProp_{h(^{-1}(M))}((n,n',0))} = name_{metaProp_M((n,n',0))}$ *and*

(2) $owner_{metaProp_{h(^{-1}(M))}((n,n',0))} = owner_{metaProp_M((n,n',0))}$

(1) $name_{metaProp_{h(h^{-1}(M))}((n,n',0))} = str(i)$

$\Rightarrow$ $\exists(n, n', i) \in Edges_{h^{-1}(M)}$

$\Rightarrow$ $name_{metaProp_M((n,n',0))} = str(i)$

(2) $name_{metaProp_{h(h^{-1}(M))}((n,n',0))} = ruleClass_{\mathfrak{G}}(rule_{h^{-1}(M)}(n))$

Link can only start in object values

$\Rightarrow$ $metaType_M(n) = (unique_{\mathfrak{G}}(r), false)$

$\Rightarrow$ $owner_{metaProp_{h(h^{-1}(M))}((n,n',0))} = ruleClass_{\mathfrak{G}}(r) = (unique_{\mathfrak{G}}(r), false)$

Proper meta-model instantiation and $metaType_M(n) = (unique_{\mathfrak{G}}(r), false)$

$\Rightarrow$ $owner_{metaProp_{h(h^{-1}(M))}((n,n',0))} = (unique_{\mathfrak{G}}(r), false)$

3.5.3 Conclusions

I showed that meta-modelling is as expressive as grammar-based language descriptions. The formal analysis theoretically allows to create meta-models from grammars, and map textual representations to corresponding models. But, this is limited to language descriptions that actually can be described by context-free grammars. Since notations defined by context-free grammars have limited complexity, there is still a gap between notations and languages. Therefore, the given grammar to meta-model mapping and respective parse tree to model mappings are a necessary fundament for notations and their descriptions, but a complete notation description requires additional constructs that describe notation features that exceed the complexity of context-free grammars. In the following sections, I use the mappings presented in this sections for notation descriptions and editors based on this descriptions. When necessary, I augment these notation descriptions with additional information about context-sensitive notation constructs.

3.6 Developing Meta-Models from Context-Free Grammars

Many existing computer languages are only defined by means of a notation. Such notations are usually described in context-free grammars. But, there is also often a demand to create meta-model-based descriptions for the languages, and separate notations from meta-models. There are two reasons: firstly, language developers want to relate two different languages with each other, which is hard to do based on concrete notations; secondly, language developers want to profit from existing meta-model-based technology, such as frameworks for model transformation or code-generation.

MOF-like meta-modelling provides four major advantages over context-free grammar-based notations, when it comes to language integration, building language tools, and defining additional language aspects. Firstly, meta-modelling allows to modularise larger language descriptions and allows to relate different language descriptions from different meta-models. Secondly, object-oriented meta-modelling allows to define abstraction hierarchies: language constructs can inherit from more abstract constructs, language constructs can be reused within the same language or among different languages. Thirdly, due to their object-oriented nature, meta-models align well with the object-oriented programming techniques used to build language tools. Finally, modularisation and object-orientation also affect the descriptions of other language aspects, which refer to meta-model elements.

In this section, I present a process to create meta-models from existing BNF grammars based on the general relations between grammars and meta-models presented in the previous section. I also present a programming framework called *More Meta-Models* (MMM) that allows to facilitate this process effectively and conveniently. Both, process and framework are evaluated through creating a meta-model for SDL based on its standardised grammar. I use SDL to exemplify the process and framework use in this section.

SDL, known as a graphical language, is only formally defined in its concrete textual syntax. SDL's lesser known textual syntax, never the less, is a fully capable replacement of its graphical syntax. SDL also defines an abstract syntax. This abstract syntax description is also provided as a context-free grammar. The abstract SDL syntax not only abandons notational artefacts like keywords, commentaries, or other special characters, it also reduces SDL to a set of core constructs. For the objectives in this section, it is assumed that stripping a concrete syntax from all notational ballast is a trivial task. Therefore, I use SDL's abstract syntax grammar to

exemplify meta-model creation from a context-free grammar.

In the previous section, I already presented a formal mapping between grammars and meta-models, allowing to create an equivalent meta-model from a grammar. In this section, I present a concrete process of creating a meta-model from an existing BNF grammar. Section 3.6.1 describes this process in general. Section 3.6.2 briefly shows how the formal grammar to meta-model mapping is utilised to generate meta-models from grammars. I am not interested in just an arbitrary grammar equivalent meta-model, but a meta-model that uses the full potential of meta-modelling. Therefore, the main concern is to enhance the automatically created meta-model. In section 3.6.3, I present techniques to manually enhance a meta-model that was automatically generated from a grammar.

3.6.1 A Process to Create Meta-Models from Given Notations Based on BNF Grammars

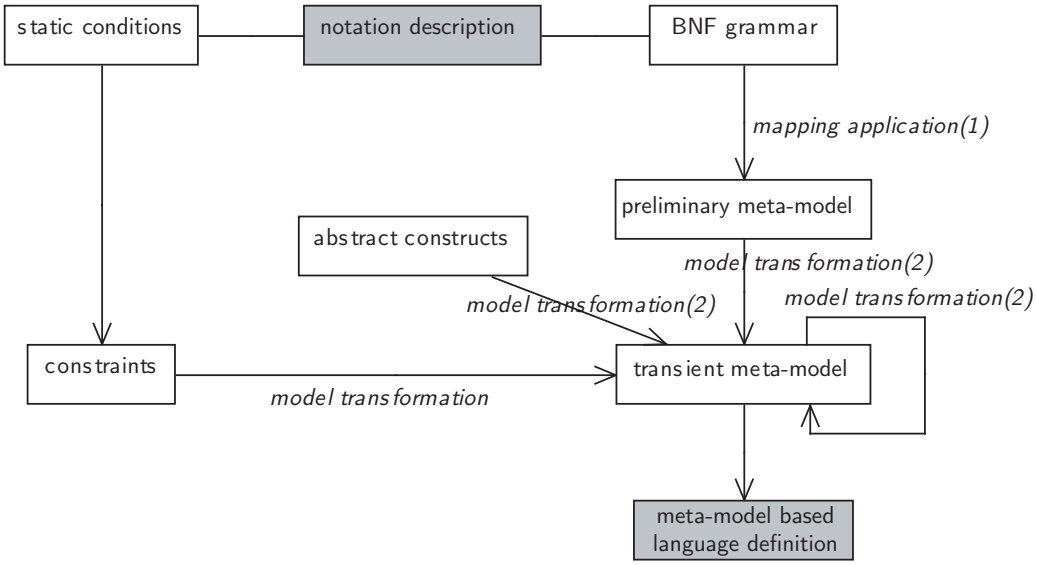


Figure 3.6: The steps involved in the presented metamodel development.

The process is illustrated in Fig. 3.6. The process starts with a notation description that contains a context-free grammar in a BNF or even EBNF format. The notation description might also contain static conditions that further limit the strings generated by the grammar. As a result of this process, you want a meta-model-based language description. This description comprises a MOF-like meta-model and OCL constraints that reference this

meta-model. The notation to meta-model development process includes two steps. First, a preliminary meta-model is automatically generated from the existing notation's BNF grammar. I require to do this step automatically, because a manual transformation from a grammar to a meta-model would cause to many errors due to human failure. The second step addresses the fact that meta-models are more expressive than grammars. In the second step the preliminary meta-model-based language description is manually revised by adding modularisation, reuse abstract language constructs, distinguish between composite and non-composite relationships, and added OCL constraints to the meta-model. In this thesis, I only addresses issues involving grammar and meta-model; I do not cover aspects of static conditions; but, a complete presentation of the method, including static conditions, can be found in [101].

3.6.2 Generating Meta-Models from a BNF Grammar

The developed MMM framework contains a tool that can generate meta-models from EBNF grammars based on the grammar to meta-model mapping given in section 3.5.1. This tools reads a EBNF grammar using a parser for EBNF grammars. During parsing, the tool creates an internal representation for the grammar. After parsing, the tool uses a transformation that realises the grammar to meta-model mapping to create the meta-model as an instance of the MOF model. This meta-model is then exported as an XMI file. This XMI file can then be used by other framework tools used to enhance the automatically generated meta-model.

3.6.3 Manually Enhancing the Automatically Generated Meta-Model

The meta-models one get by applying the described mapping from a BNF grammar can be called primitive at best. The reason is that those primitive meta-models can only be as expressive as the original grammar is. Several problems exist in the preliminary meta-models.

Those preliminary meta-models suffer from the same drawbacks as the original grammars: they do not include any abstraction relationships between language constructs; they do not distinguish between compositional and non-compositional associations (grammars define trees, meta-models graphs)[6];

[6]The terms *compositional* and *non compositional*, hereby, refer to structural features that imply and not imply exclusive ownership. The corresponding structural features can carry other model elements as values. One model element used as a value in a composi-

they are not modularised. I want to have a detailed look on these problems and get into a few examples, before I elaborate on solutions to these problems.

Typically used context-free grammars formalisms do not allow grammars that define language constructs on different abstraction levels. One example are the structural type constructs in SDL, namely *Agent type definitions* and *Composite-state type definitions*. These two constructs are generalizable, they can be instantiated, they are namespaces for a number of other SDL constructs, and so forth. But these abstract constructs are specified separately for both elements, instead of being defined once and then reused. Especially being a namespace is a property that even more SDL constructs like procedures and packages share. Furthermore, namespace is an abstract concept used by many other languages; UML or most programming languages are examples.

```
Procedure_definition :: Procedure_name
    Procedure_formal_parameter*
    [ Result ]
    Procedure_graph;

Procedure_name = Name;

Procedure_formal_parameter = In_parameter
    | Inout_parameter
    | Out_parameter;
```

Figure 3.7: An excerpt from the SDL grammar.

In grammars there is no difference between attributes or interclass relationships. Furthermore, grammars do not distinguish between classes and data types. In the grammar example in Fig. 3.7, both *Procedure_definition* and *Procedure_name* are described by a symbol. Therefore both constructs are modelled by a class in the generated meta-model. Of course, that is bad meta-modelling technique. The name of a procedure should rather be modelled through a string attribute. The same problem lays in the modelling

tional feature of another element becomes part of the other element. The part is called a *component* of the owning element; the owning element is called *container* respectively. Each model element can be the component of one container only, i.e. it can be a value in the structural features of one single model element at best. The component-container relationship is called *composition*. Cycles in composition are not allowed; composition forms trees within models. The roots of these trees are model elements without container; the leafs are elements without components. [117]

of associations. Meta-modelling concepts like navigability, aggregation, etc. cannot be stated in grammars, and so they are not used in the generated meta-model, even if their use would be appropriate.

In grammars each relationship between language elements is of compositional nature. SDL constructs like *identifier* and *qualifier* realise non compositional references between language elements in an indirect way. In a text-based language they are needed to identify objects. To do so, they represent a logical relation between the object definition and its use, like the definition of a variable and its use in an expression. In a meta-model and its model instances these constructs are not necessary. These relations can be modelled by associations and their instances, called links. In other words these are constructs that already exist as meta-meta-constructs and have not to be redefined. Of course, *identifier* and *qualifier* are needed in concrete model notations to represent those relations, but they serve no purpose in a meta-model-based language description.

In the following, I further describe the idea of abstract language constructs and which constructs are interesting for SDL. After that, I look at ways to integrate these abstract constructs with the generated meta-model, and how to address the other issues by transforming the generated meta-model.

Abstract Construct Definitions

Firstly, I present the abstract construct definitions that were used for the SDL meta-model. There are different levels of abstractions. Some constructs are so general that they are used in virtually every object-oriented language, others are more specific and may only be reused among related languages or only within one language.

Fig. 3.8 shows the abstract structure constructs used by most object-oriented languages. A detailed explanation and documented development of that model can be found in [101]. These constructs are not only relevant to SDL, they are also used in the meta-models of UML and the meta-meta-model of MOF.

But even more abstract constructs could be obtained from SDL's notation itself. Even if they may turn out to be more specific, perhaps distinctive to SDL, they still allow a more compact and therefore easier to understand and easier to use meta-model. Fig. 3.9 presents the additional abstract constructs that could be identified in the SDL notation. A few concrete meta-model classes, those that are marked grey, are shown too. A few remarks:

- Many SDL constructs reference a *body* of some sort. Procedures for example must contain a state automaton defining their behaviour. This

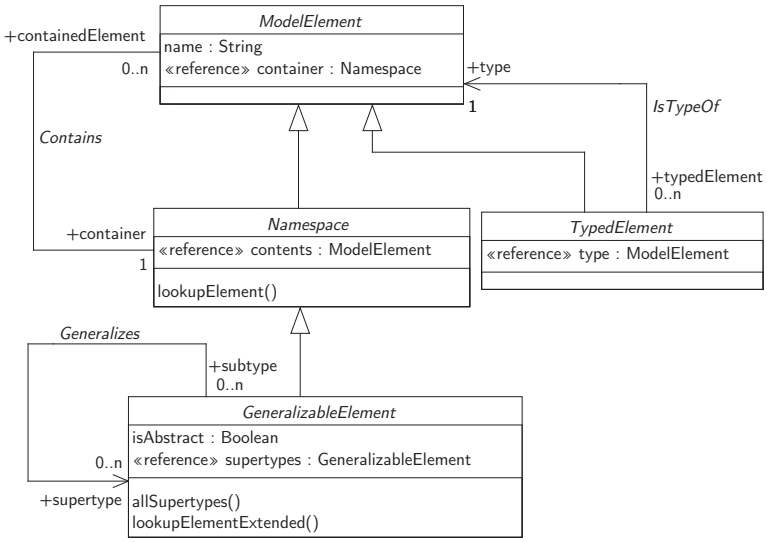


Figure 3.8: Abstract constructs.

state automaton is referred to as a *body*. The same is true for process typed *Agents* or the bodies in *Composite-state types*. To respect the varying nature of bodies, they are modelled to be the most abstract construct: *ModelElement*.

- Parameters are used by a variety of SDL constructs. *Agent types, Procedures, Composite-state types* have parameters. Even if *Procedure* uses a special form of parameter, the parameter itself is a typed element in any case.

- In SDL two type constructs coexist. A type is something that describes a set of instances or values. In SDL a type can on the one hand be a data type, like a *Signal definition* or a primitive data type and on the other hand a type can be a structure type like *Agent type* or *Composite-state type*.

- Structure types can be instantiated and are generalizable, parametrized types. Therefore structure types are a combination of the generalizable construct, parametrized construct and the body possession construct.

As a reminder: one may criticize that the displayed model allows unwanted instances, that procedure for example may contain a non-procedure parameter, or a structure-typed element may reference a data type. Obviously some restrictions have to be added to the model. Actually these

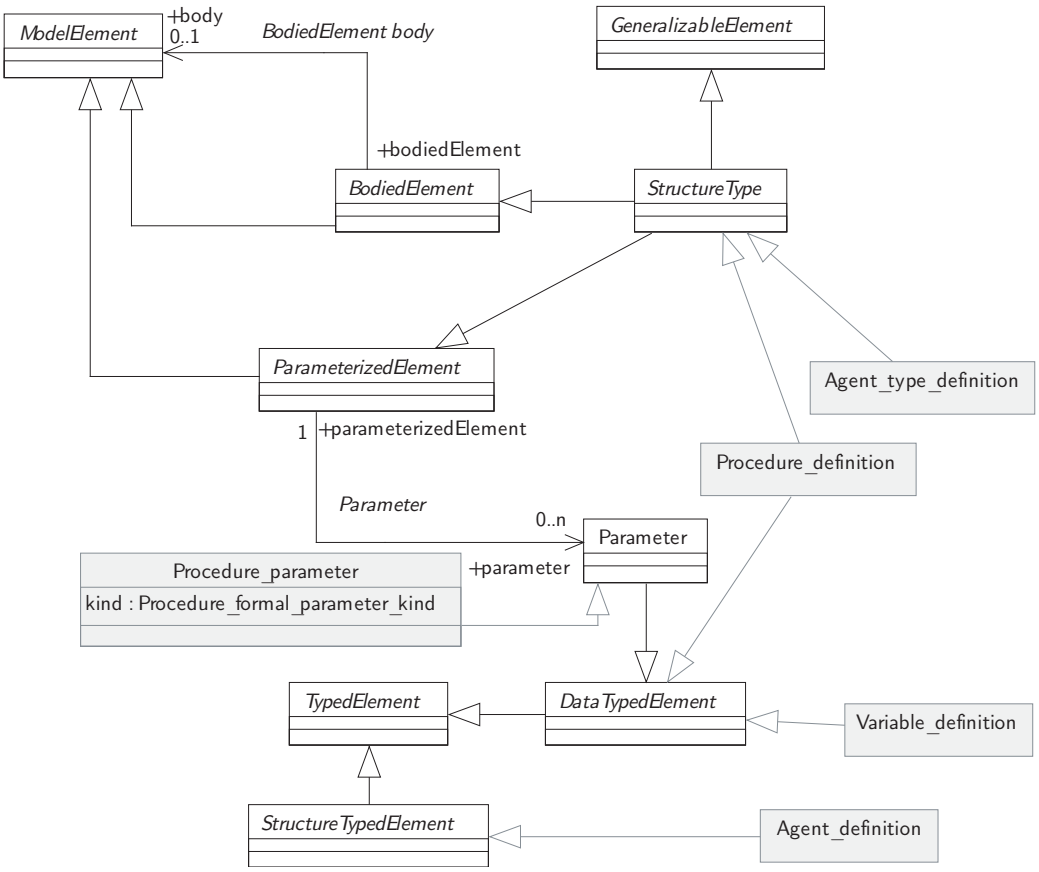


Figure 3.9: Abstract SDL constructs.

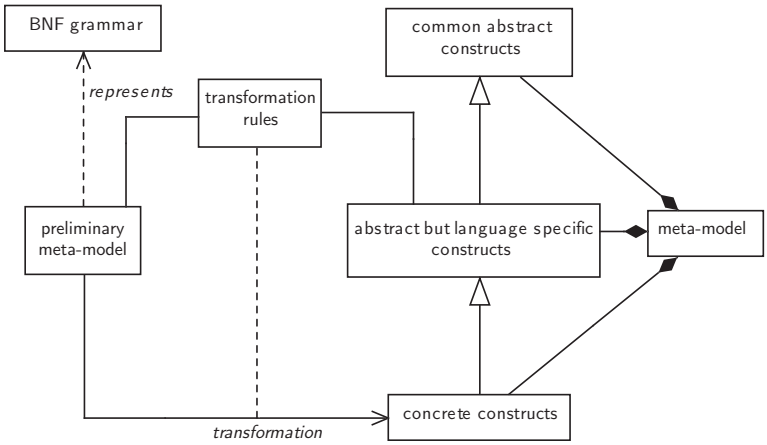


Figure 3.10: Transforming the preliminary meta-model.

constraints are considered static conditions and are not covered by this thesis, but [101] addresses that matter by using OCL to further limit the set of possible meta-model instances.

Transforming the Preliminary Meta-Model

Now, you have sets of abstract meta-model elements and a generated preliminary meta-model. But how to combine these model elements to form a single meta-model? How to further manually enhance the meta-model? To merge the generated meta-model with the abstract meta-model parts, two things must be realized: firstly, the concrete constructs must be marked as specializations of the introduced abstract constructs. Secondly, features and rudiments of concrete elements that are already defined or realized by the corresponding abstract model element must be removed. To accomplish this task, I use model transformation. I further use model transformation to resolve any other grammar related issue in the meta-model, such as replacing composite relations with attributes, replace identifier-based references with real non-compositional relations, introduce package structures, and so on.

Fig. 3.10 shows the basic idea of this transformation. All the transformations needed, require information from a language expert. I call this information *transformation rules*. A language developer can express the relation between concrete constructs and abstract constructs and how they refines the abstract constructs in this transformation rules. Transformation rules, can furthermore describe how the other described problems are to be solved.

With this transformation rules the transformation itself can be done au-

```
agentTypeDefinition = new SdlStructureTypeAdaptor(
        "Agent_type_definition");
agentTypeDefinition.addSupertypeType("Agent_type_defintion");
agentTypeDefinition.addBodyType("State_machine_definition");
agentTypeDefinition.addParameterType("Agent_formalparameter");
agentTypeDefinition.addContainedType("Agent_type_definition");
agentTypeDefinition.addContainedType("Procedure_definition");
agentTypeDefinition.addContainedType("Agent_definition");
    ...
```

Figure 3.11: An example taken from the semantic mapping used for the SDL metamodel.

tomatically. The MMM framework therefore provides a programming library that allows language developers to define the necessary transformation rules as a simple program. This library works as a small transformation language.[7]

As an example, Fig. 3.11 presents a part of the transformation rules used for the SDL meta-model. The first line assigns *Agent_type_definition* to be a specialization of the abstract class *StructureType*. The third line maps agent type definition's association with itself to be a specialization of the generalization association introduced by one of agent type definition's new super meta-classes: *GeneralizableElement*. Line four maps agent type definition's association with state machine definition to be a specialization of the body association introduced by another new super type of agent type definition: *BodiedElement*. Line five refines the inherited features of the abstract class *ParameterizedElement*, the other lines refine the inherited features of the abstract construct *Namespace*. For every abstract construct an adaptor class was written. *SdlStructureTypeAdaptor* is such an adaptor. The inheritance hierarchy of the adaptors are aligned to the hierarchy formed by the corresponding abstract constructs. Thus the super types of *SdlStructureTypeAdaptor* are *GeneralizableElementAdaptor*, *ParameterizedElementAdaptor*, *BodiedElementAdaptor*, *NamespaceAdator* and *ModelElementAdator*.

For every abstract construct class, an adaptor class was written, for every concrete class an adaptor instance is created during transformation. The constructor is used to map the concrete class to the abstract construct class. This means: the constructor introduces a new generalization relation between the abstract construct class and the concrete construct class that is provided

[7]The use of libraries as languages is known as *internal domain specific language* [34]; internal because the language's concrete syntax is expressed in an existing host programming language.

through the constructor's parameter.

For every abstract association or attribute a method was written. The method is owned by the adaptor class for the abstract construct class, that the association or attribute is originated in. For every concrete association a method call is used. This Java method call maps the concrete association to the corresponding abstract association. Therefore the Java method deletes the old concrete association, originally generated through the grammar to meta-model generator, and replaces it with a refinement of the abstract association. This refinement is done through the addition of a constraint. The concrete association is identified by taking the concrete class for one end and taking the class provided through the Java method's parameter for the other end.

For example, look at line three of the transformation rules example in Fig. 3.11. Originated in SDL's abstract grammar, a concrete association between *Agent_type_definition* and *Agent_type_definition* exists in the primitive meta-model. This association refers to the inheritance relationship between two agent types. Line three maps the abstract association *Generalizes* of *StructureType*'s and therefore *Agent_type_definition*'s meta- superclass *GeneralizableElement*. The invoked Java method removes the original concrete association and replaces it by a constraint that restricts the abstract inherited *Generalizes* association to allow only links between two instances of *Agent_type_definition*. The mapping of *StructureType* associations then continues for the abstract associations *Contains* and *ElementBody*.

This way the transformation works as a chain of commands that transforms the meta-model according to the semantics given by the transformation rules. After transformation, all concrete constructs are specializations of abstract ones, all concrete associations and rudiments have been removed and replaced by constraints that restrict abstract associations or attributes. The only things left are a few concrete constructs for which no appropriate abstract constructs could yet be identified. For these constructs some additional transformation rules are necessary that create attributes from associations and that replace identifier-based relationships with non-compositional associations.

3.7 Creating Text Editors and Textual Model Editors

Modern integrated development environments for programming languages have accustomed software developers to *language specific text editors* with

capabilities that increase the programmers productivity. Extensive knowledge about language syntax and semantics programmed into these editors, allows editors to offer assistance to the editor user. Such editors add additional visual information such as highlighting keywords and literals, marking occurrences of the same identifier, and visualising distinct properties with different colours. Editors show an outline of the edited content to allow easier navigation and furthermore allow to navigate between identifier and identified element with a single click. Assistance is also provided in the form of error and warning markers that give feedback about the statical correctness of the written program. Furthermore, modern editors can present the user with a list of meaningful continuations at the current cursor position. This is known as *content assist* or *code completion*. In general, those language specific text editors allow the user to concentrate on the programming task, while concerns of syntactical and statical correctness, recalling identifier names, and navigating through complex program structures are handled by the editor.

Unfortunately, development of such editors is extensive and therefore has only been done for popular programming languages like Java. In this section, I want to develop such feature rich text editors with Domain Specific Modeling (DSM) [51]. Language developers do not program editors directly, but describe them, and automatically create editors from these descriptions. DSM for language tools relies on meta-languages and meta-tools: language developers use meta-languages to describe an editor, and meta-tools generate the editor from the description.

This section is about a framework for the DSM of language specific text editors. This framework is called *Textual Editing Framework* (TEF) [104], and serves as a prototypical implementation and experimental platform to prove the applicability of DSM to textual model editors. I use TEF to experiment with and analyse the techniques that are presented in this section. TEF features a meta-language based on the relationship between context-free grammars and meta-models. This meta-language allows to define notations based on context-free grammars, and allows to relate this notation description to a meta-model. The editor, generated from such a description, uses a method called *background parsing*, which allows the editor to create a model from the textual representation that the user edits. This automatically created model is not only the current model edited in this editor, it is also used by the editor to realise all language specific editor features. The presented framework is based on the *eclipse* platform, and use the *Eclipse Modeling Framework* as a MOF-like meta-modelling framework.

I distinguish two different kinds of textual editors depending on what kind of files they can edit. The first kind are simple *text editors*. They allow

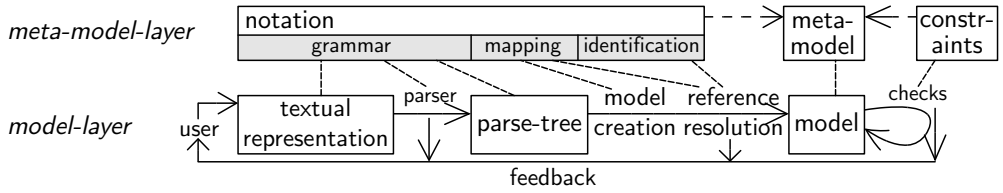


Figure 3.12: The background parsing strategy and involved artefacts.

the editing of text files, files that contain a textual representation of a model. The second kind are *textual model editors*. These edit model files. In the case of MOF-like modelling, these are usually files containing models represented in XMI. These editors allow to edit a textual model representation, but store the model itself. Both editor kinds appear similar to the user, they both offer the same features, and both create models from the edited representation to realise these features. The only difference between these two kinds of editors is the way they store models or model representations.

With the development efficiency of DSM, the presented framework makes the development of language specific textual editors for smaller languages practical. This includes text editors and textual model editors for domain specific languages. This also includes textual model editors for small textual parts of otherwise graphically represented models that I call *embedded editors* (refer to Fig. 3.20).

This section is structured as follows. In the beginning, I elaborate on background parsing in 3.7.1, before I look at the meta-language used to describe background parsing editors in subsection 3.7.2. In subsection 3.7.3, I look at the technical details of realising the presented meta-language and the corresponding meta-tools, which constitute the TEF framework. The following subsection 3.7.4 discusses content assist: the description and realisation of it. In subsection 3.7.5, you see how text editors can be integrated into graphical editors to combine graphical and textual model editing. The next subsection discusses alternative textual editing techniques and frameworks 3.7.6.

3.7.1 Textual Model Editing with Background Parsing

Background parsing is a strategy to realise textual editors. Each textual model editor has to create a model from the textual representation that the user edits. Background parsing allows the user to edit the textual representation as in any other text editor. The editor creates a model from the edited text in cyclic intervals. This relies on parsing based on a

context-free grammar. Background parsing happens without user awareness. Background parsing has been used in many text editors and frameworks [125, 69, 48, 59, 13].

Background parsing is controlled by a language and notation descriptions, including a context-free grammar, a meta-model, and a mappings from grammar elements to elements of the meta-model. Furthermore, the description contains information about notation features that go beyond the expressiveness of context-free grammars. The last part includes information about identifier resolution and information needed to control editor features like syntax highlighting or content assist. These descriptions are basically used as follows: The notation's grammar defines a set of syntactical correct textual representations: it defines the notation language. The grammar to meta-model mappings and identifier resolution information identify the model corresponding to a given representation: these items define a mapping between the notation language and the notated language.

Background parsing is a circular process of four steps. First, the user edits text as in a regular text editor. Second, the inserted text is parsed according to the notation's grammar. Third, a model is created from the resulting parse tree. Finally, language constraints are used to check the model. Errors in representation or resulting model can arise in all steps and are reported back to the user as annotations in the edited text. Otherwise, the user is unaware of the parsing process, and continuous repetition gives the impression that the user edits the model directly. As opposed to other editing strategies, background parsing does not change the currently edited model. Instead, it creates a completely new model in each repetition. The background parsing process and involved descriptions are illustrated in Fig. 3.12.

I want to further illustrate background parsing and the involved descriptions with the following example. Fig. 3.13 shows a meta-model for a simple expression language in the top-right corner. Below this meta-model, you see a model, an instance of the meta-model, that represents the expression $foo(n) = (n + 2) * n + 1$. On the left side of this figure, you see a grammar that could be used to define a notation for that expression language. The rules in this context-free grammar can be used to create the parse tree below, which is a parse tree for the string $foo(n) = (n + 2) * n + 1$.

The example parse tree and the example model are very similar. Basically, you can map symbols and terminals to objects and their attributes, and you can map child-of relations between nodes to corresponding links. However, there are important differences. There are some links in the model that are not represented in the parse tree directly. These are links between instances of *VariableExpression* and *Variable*. The fundamental difference between meta-models and context-free grammars is that meta-models describe graphs,

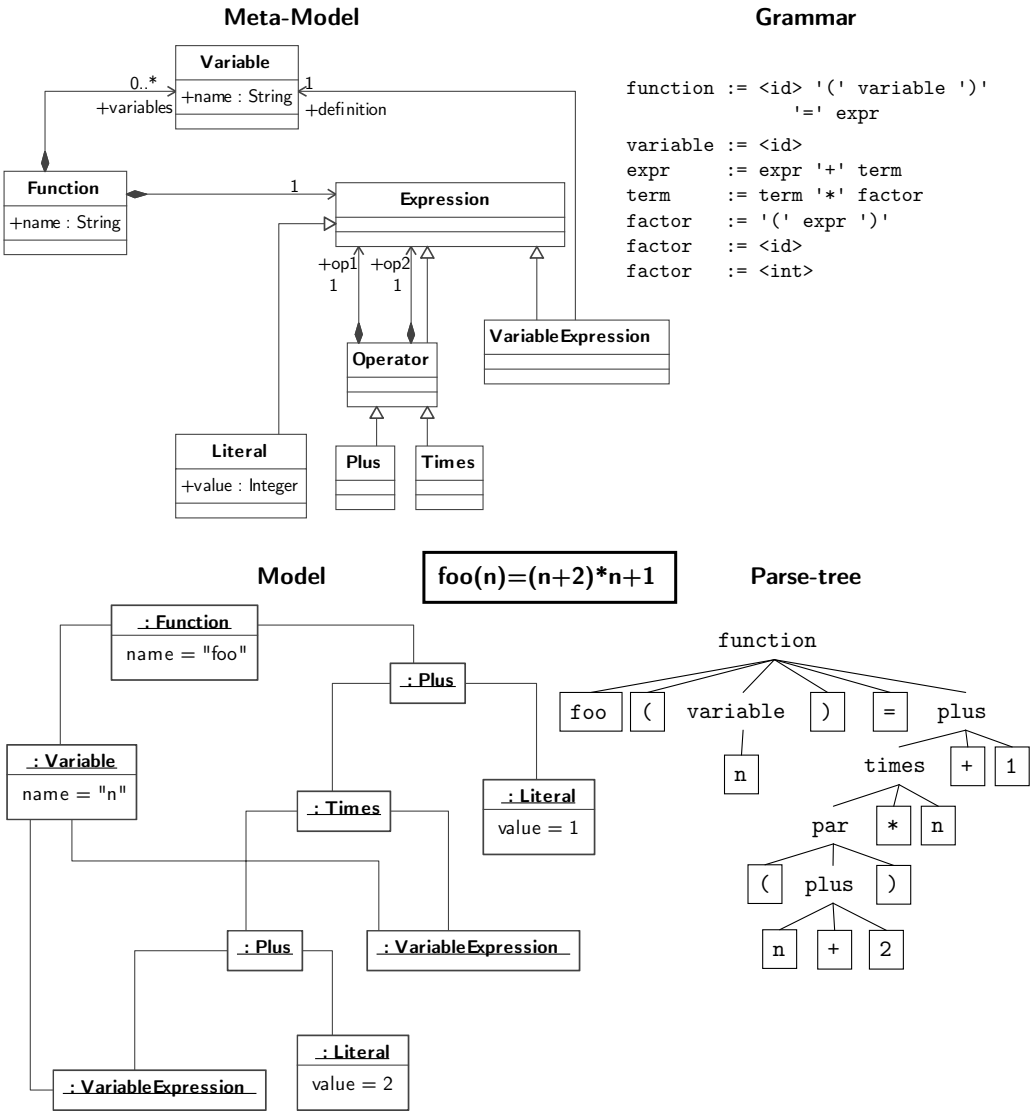


Figure 3.13: The differences between meta-models and grammars exemplified based on a simple expression language.

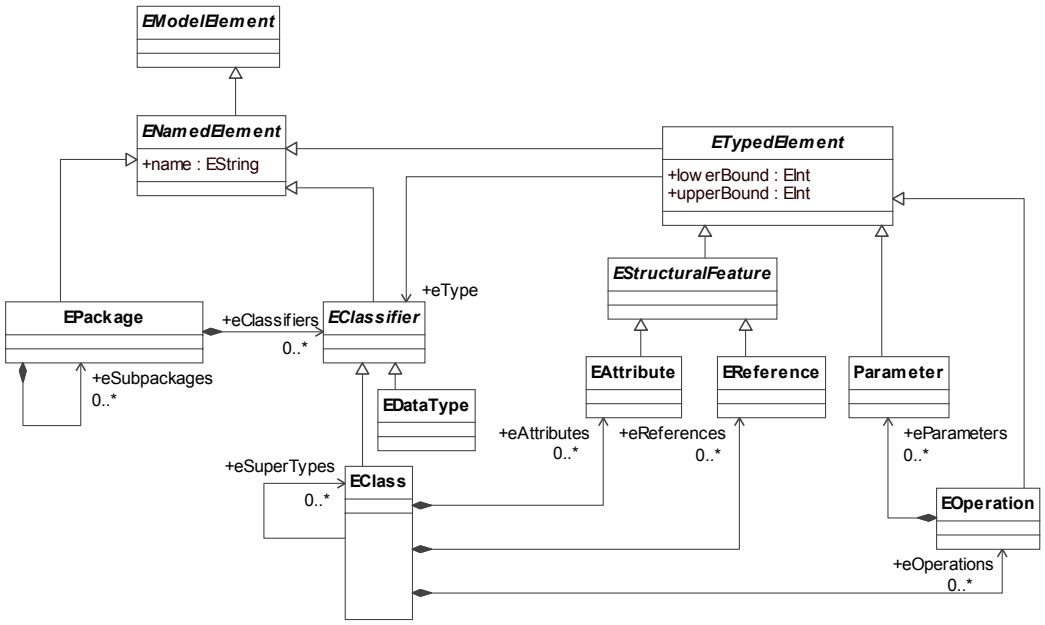


Figure 3.14: The Ecore meta-model simplified for the example.

while grammars describe trees. Therefore, this link (which causes a circle in the model graph) has to be represented indirectly within the parse tree. This is a typical problem for notations defined with context-free grammars: they usually use some form of identifier to describe a reference between the variable definition and a use of that variable. To create a model from a parse tree, creation has to include some form of *identifier resolution*.

With the given meta-model, grammar, and additional grammar to meta-model mappings, including information about identifier resolution. An editor using background parsing can create the parse tree automatically and can create the model from this parse tree automatically. In the next section, I take a closer look on the necessary descriptions and how they work together.

3.7.2 Notation Language and Notation Semantics

In this section, the description artefacts for textual notations are described as they are used as input for the Textual Editing Framework (TEF). These descriptions are combined in a description language for textual notations. This meta-language is part of TEF, and a description written in that language in combination with an corresponding EMF meta-model allows TEF to create either a textual model editor or a text editor.

The concepts in this section are exemplified with a textual notation for

Ecore models. Ecore is the meta-modelling language of EMF and provides MOF-like meta-modelling constructs. The notation that I use, uses a Java/C++ like syntax to represent Ecore models. Fig. 3.14 shows a simplified version of the Ecore meta-model. The simplifications are only needed to create a clearer example, they do not present technical limitations. The example notation given in this section, is a notation for this meta-model. The real notation that was developed, is, of course, a notation for the complete Ecore meta-model. Fig. 3.16 shows an example representation written in that Ecore notation.

Grammar

To define a set of syntactically valid textual representations, you use context-free grammars. Fig. 3.15 shows such a context-free grammar for the Ecore example notation. An example string that can be generated with this grammar is the example representation of an Ecore model in Fig. 3.16. The shown grammar is a context-free grammar with EBNF elements. All string literals function as fixed terminals; all capitalized symbols are morphem classes for integers or identifiers; everything else are non-terminals.

The specific grammar format, used in this example, is the specific grammar format that is used in TEF's notation description meta-language. The grammar is used as input for a parser-generator[8]. TEF facilitates the parser-generator to create parsers. These parsers allow to syntactically analyse textual representations and create parse trees from successful parsings of textual representations.

Grammar to Meta-Model Mapping

To control the creation of models from parse trees within an editor, the notation description has to provide mappings between grammar elements and meta-model elements. These mappings allow the editor to instantiate meta-model classes from parse tree nodes.

In Fig. 3.17 the grammar from the previous subsection (Fig. 3.15) is extended. The bold printed elements relate grammar elements to meta-model classes and features. If such a meta-model relation is attached to the left-hand-side of a rule, the rule is related to a meta-model class; if attached to a right-hand-side symbol, the right-hand-side rule part is related

[8]For TEF the parser-generator (or compiler compiler) Runtime Compiler Compiler (RCC)[90] was used. This programming library allows to generate parsers from context-free grammars at runtime, and was chosen because it can create parsers within a running eclipse instance, without the need to restart the whole platform.

```
Package ->
    "package" IDENTIFIER "{"
        (PackageContents)*
    "}";

PackageContents -> Package;
PackageContents -> Class;
PackageContents -> DataType;

Class -> (AbstractSwitch)? (InterfaceSwitch)?
    "class" IDENTIFIER (SuperClasses)? "{"
        (ClassContents)*
    "}";

AbstractSwitch -> "abstract";
InterfaceSwitch -> "interface";

SuperClasses -> "extends" ClassRef ("," ClassRef)*;
ClassRef -> IDENTIFIER;

DataType ->
    "datatype" IDENTIFIER ("["INSTANCE_CLASS_NAME"]")? ";";

ClassContents -> "attribute" Attribute;
ClassContents -> "reference" Reference;
ClassContents -> Operation;

Attribute -> IDENTIFIER ":" TypeExpr Multiplicity;

TypeExpr -> TypeRef;

Reference -> IDENTIFIER ":" ClassRef Multiplicity;

Operation -> IDENTIFIER "(" (Parameter ("," Parameter)*)? ")"
    ":" OperationReturn;

OperationReturn -> TypeRef;
OperationReturn -> "void";
```

Figure 3.15: A context-free grammar in EBNF for the Ecore notation.

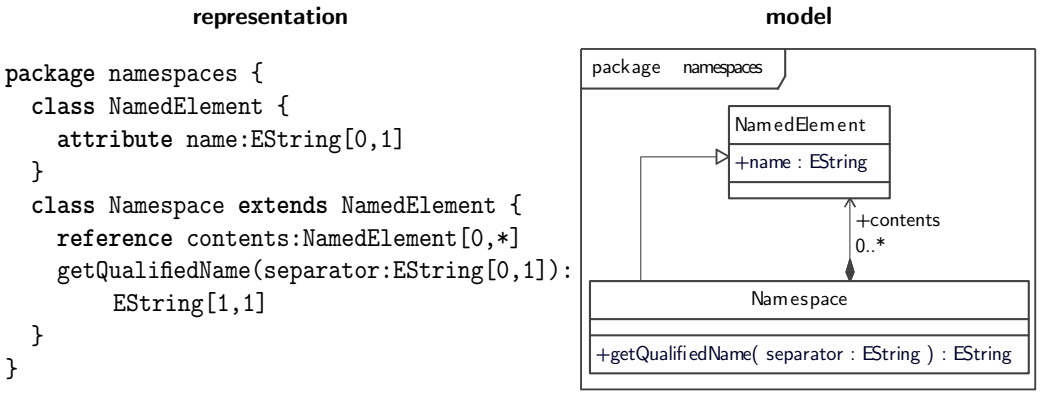


Figure 3.16: An example representation in the Ecore notation and the model it represents.

to a feature. I distinguish between different kinds of meta-model relations: *element* relations relate to classes, *composite* relations relate to attributes or other compositional structural features, and *reference* relations relate to non compositional structural features.

Creating Models from Parse Trees

The result of parsing a textual representation is a syntax-tree or *parse tree*. Fig. 3.18 shows such a parse tree. This parse tree is the result of parsing the shown example representation with the grammar previously introduced in Fig. 3.15. Each node in a parse tree is an instance of the grammar rule that was used to produce the node. To create a model from a parse tree, editors perform two depth-first traversals on the tree.

In the first run, editors use the element relations attached to a node's rule to create an instance of the corresponding meta-model class. This instance becomes the value represented by this node. It also serves as the context for traversing the children of this node. Morphems create corresponding primitive values, e.g. integers or strings. During the same traversal, I use the composite relations to add the values created by the respective child nodes to the referenced features in the actual context object. With this technique, the parse tree implies composition between model elements. Furthermore, the compositional design of the meta-model must match the alignment of corresponding constructs in the textual notation.

In the second traversal (also called *identifier resolution*), the editor goes through parse tree and model simultaneously. Now, it uses all reference relations to add corresponding values to all non compositional structural features. This time, it does not use the child nodes' values directly, but uses

```
syntax(Package) "models/Ecore.ecore" {
    Package:element(EPackage) ->
        "package" IDENTIFIER:composite(name="unnamed") "{"
            (PackageContents)*
        "}";

    PackageContents -> Package:composite(eSubpackages);
    PackageContents -> Class:composite(eClassifiers);
    PackageContents -> DataType:composite(eClassifiers);

    Class:element(EClass) ->
        (AbstractSwitch:composite(abstract))?
        (InterfaceSwitch:composite(interface))?
        "class" IDENTIFIER:composite(name="unnamed")
            (SuperClasses)? "{"
                (ClassContents)*
        "}";

    AbstractSwitch:constant("true":Boolean) -> "abstract";
    InterfaceSwitch:constant("true":Boolean) -> "interface";

    SuperClasses -> "extends"  ClassRef:reference(eSuperTypes)
        ","  ClassRef:reference(eSuperTypes))*;
    ClassRef:element(EClass) ->
        IDENTIFIER:composite(name="unnamed");

    DataType:element(EDataType) ->
         "datatype" IDENTIFIER:composite(name="unnamed")
             ("["INSTANCE_CLASS_NAME
                :composite(instanceTypeName)"]")? ";";
    ...
    Attribute:element(EAttribute) ->
        IDENTIFIER:composite(name="unnamed")
        ":" TypeExpr Multiplicity;
    Reference:element(EReference) ->
        IDENTIFIER:composite(name="unnamed") ":"
        ClassRef:reference(eType) Multiplicity;
    ...
}
```

Figure 3.17: The Ecore notation description with meta-model mappings.

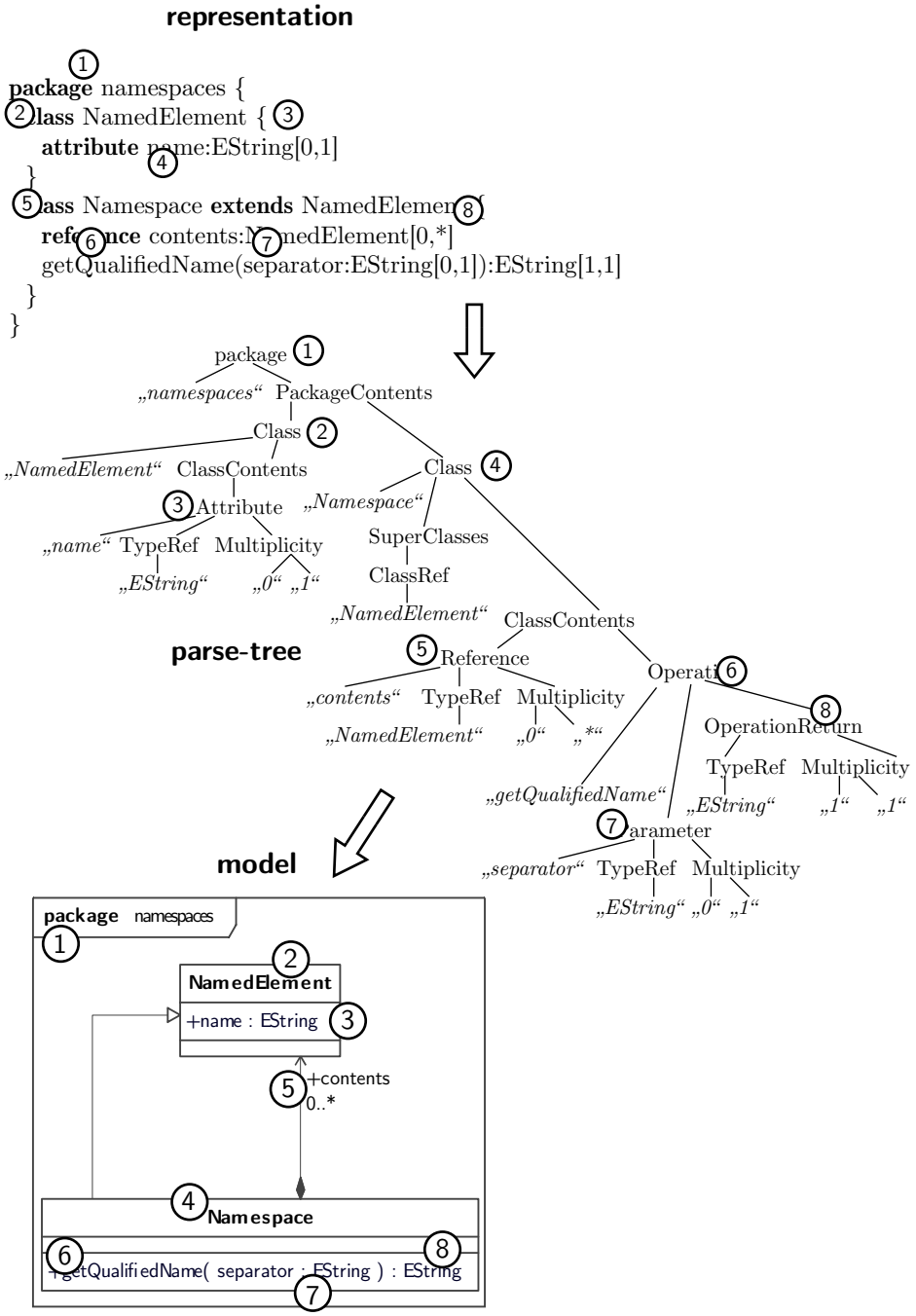


Figure 3.18: A representation, parse tree, and model based on the example Ecore notation.

the child nodes' values as identifiers to resolve the corresponding referenced elements. Since all model elements were created in the first traversal, the referenced model elements must already exist in the model. What identifiers are and how editors resolve them is explained in the next subsection.

Identity and Reference Resolution

To understand how editors add feature values for references in the second traversal, you have to understand identification of model elements.

Definition 33 (identity) *The* identity *of a model element is a value that uniquely identifies this element within a model.*

The notation of a language defines a function that assigns each element of a language instance an identity. Example identities can be the model element itself, the name of the model element, or more complex constructs like fully qualified names.

Definition 34 (identifier) *An* identifier *is a value used to identify a model element based on the element's identity.*

In simple languages, identifiers can often be used directly to identify model elements: if an identifier and the identity of a model element are the same, this identifier identifies this model element. The notations of many languages, however, require more complex identification, including name spaces, name hiding, imports, etc. In those cases, an identifier depends on the context it is used in. The language's notation must define a function that assigns a set of possible global identifiers to an identifier and its context. These global identifiers are then used to find a model element with an identity that matches on of those global identifiers.

Identities and identifiers are notation-specific and might vary for model elements of different meta classes. Textual model editing frameworks can only provide a simple default identification mechanism, i.e. based on a simple identity derived from a model element's meta-class and possible *name* attribute (or other so-called key attributes). TEF and other frameworks allow to customize this simple behaviour. Technically, this customization is also part of the notation description. Since no specific description mechanisms for identification could be found for existing textual editing frameworks yet, this part of a notation description has usually to be programmed within the used textual editing framework. TEF provides a programming interface that allow language developers to provide a notation specific identification as a set of Java classes. [9]

[9] This is a typical practice of DSM [51]: for very specific purposes, description languages allow to leave the realm of their abstract description constructs, and allow language users

Checking the Model

The created model can be analysed. This can be done for two different purposes: firstly, finding errors and potentially dangerous conditions; secondly, derive additional properties or documentation for model elements. For both purposes, queries over the model can be used: either to check a condition (boolean query) or to derive information from the model (arbitrary typed query or string query). Based on given queries as part of a notation, you can automatically derive annotations and text markers in an editor that show this information. Queries can either be programmed, or described in specific query languages. TEF allows to either program constraints or to write OCL constraints as part of the used meta-model.

Model Changes and Pretty Printing

In the case of textual model editors, the stored artefact contains the model and not the representation. This allows that tools other than the textual model editors create or change this edited model. There are, among others, three scenarios: firstly, the user saves the edited text as a model, changes the model with another tool and then continues editing the model in its textual representation; secondly, the user changes the model during editing, e.g. using refactorings on the model; thirdly, the model is changed from outside the editor by other tools and users.

In each scenario, you need to create a textual representation from an existing model. The creation of a textual representation for a model is also called *pretty printing*. To create a textual representation, an editor has to reverse the model creating and parsing process: it creates a parse tree from the edited model element and pretty prints this tree.

Creating a parse tree To create a parse tree, the editor traverses the model along its composition. For each model element, the editor can determine a set of suitable grammar rules based on the element's meta-class, the values of its features, and the grammar to meta-model mapping. By using a back-tracking strategy, the editor can determine one or more possible parse trees for a model. Notations that provide different possibilities to represent a single language construct in different ways, will lead to multiple possible parse trees for the same model. Frameworks can give language developers the possibility to prioritise grammar rules accordingly. If the editor cannot determine a parse tree for a model, meta-model, grammar, and grammar to meta-model mapping, these descriptions contain flaws or inconsistencies. For

---

to add description written in general purpose programming languages.

example, a model element name might be optional as defined by the meta-model, but required by the notation: at some point in the model traversal, all possible grammar rules for the element would require the name of the element, but the name cannot be obtained. As future work, one could examine whether each model is pretty-printable in a given notation or not.

Pretty printing the parse tree Pretty printing a parse tree is basically straight-forward. The only problem are the white-spaces between tokens. A human readable textual representation needs reasonable white-spaces, i.e. *layout information*, between these tokens. This is a problem with two possible solutions. Firstly, white-spaces originally created by editor users can be stored within the model. Secondly, editors can create white-spaces automatically. Disadvantages for storing layout information are that the layout information has to be provided by editor users, and model and meta-model have to be extended with layout information elements. The advantage is that user layouts and all information that users express within layouts (also known as secondary notation [84]) are preserved. The second solution has complementary advantages and disadvantages.

The automatic generation of white-spaces is the only solution that allows pretty printing for models that are not created by humans, and it is also the most generally applicable solution, since it does not require any pre-existing layout information. Therefore, automatic generation of white-spaces was integrated into TEF, and I want to take a closer look.

Automatic layout of textual representations requires white-space clues as part of the textual notation description. For automatic layouts *white-space roles* can be used. Language developers add white-space roles as symbols to grammar rules. A white-space role determines the *role* that the separation between two tokens plays. White-space roles are dynamically instantiated with actual white-spaces, when text is created from a model. A component called *layout manager* defines possible white-space roles and is used to instantiate white-space roles. It creates white-spaces for white-space roles in the order they appear within the created textual model representation. The layout manager can instantiate the same white-space role differently, depending on the context the white-space role is used in.

An example: A layout manager for block-layouts as used in programming languages, supports the roles *space*, *empty*, *statement*, *blockstart*, *blockend*, *indent*. This manager instantiates each *space* with a space and each *empty* with an empty string. But, if the manager detects that a single line of code becomes too long to be readable, the layout manager can also instatiate both roles with a return followed by a proper identation. The manager uses the

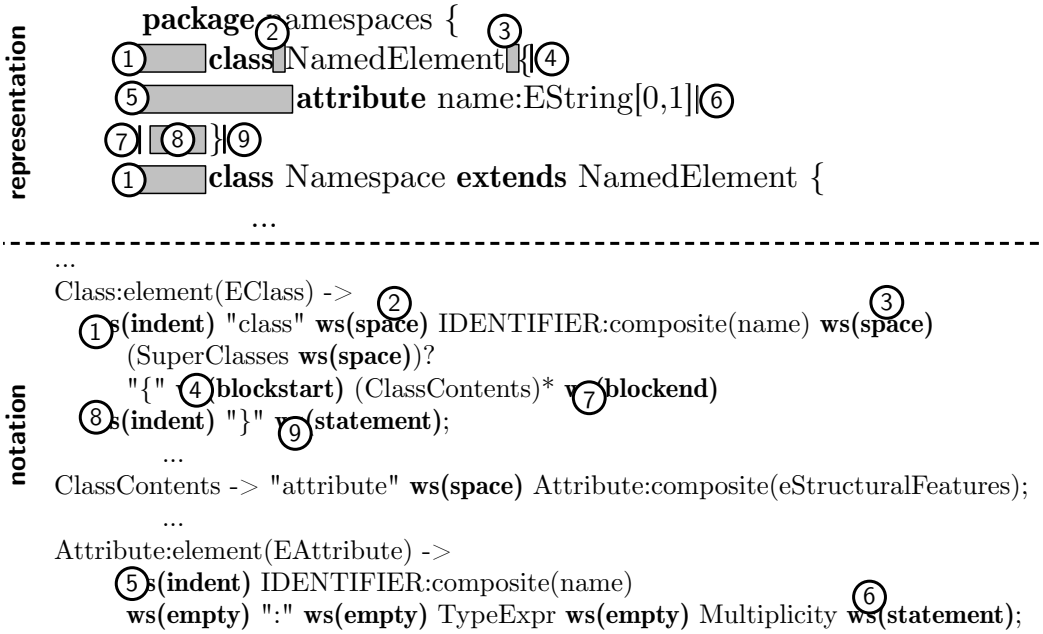


Figure 3.19: An example text with white-spaces for an example notation with white-space roles.

blockstart and *blockend* roles to determine how to instantiate an *indent*. It increases the indentation size when a *blockstart* is instantiated, and decreases it, when a *blockend* is instatiated. The *statement* role is instantiated with a return followed by an identation in the actual indentation depth. Fig. 3.19 shows an example representation and corresponding notation with white-space roles for the Ecore language based on the block-layout manager.

3.7.3 Building Textual Model Editors – The Textual Editing Framework

You now have an understanding about background parsing and how background parsing can be described in textual notations. I already explained the different description aspects and how background parsing editors can facilitate the different descriptions, but never-less, I use this section to take a deeper look at TEF and how TEF creates editors from notation descriptions.

TEF provides a notation description meta-language. You have seen examples written in that language, defining a notation for the Ecore language. TEF's notation description language allows to specify an EBNF grammar, grammar to meta-model mappings, and white-space roles for automatic layout. The parser generator used by TEF allows only LALR(1) grammars.

This is not a general limitation; a similar meta-tool could also be implemented based on a more expressive context-free grammar formalism. TEF also provides an extendible API that allows language developers to program a language-specific (meta-model-dependent) identification mechanism, to program meta-model constraints, and language specific content assists (see 3.7.4).

TEF follows a DSM paradigm. This means a language developer only provides a notation description written in TEF's notation description language, and TEF automatically creates the textual model editor or text editor. Language developers can enhance the generated editor afterwards. Examples for such enhancements are, for example, a specific identification mechanism, programmed constraints, or specialised content assist behaviour.

TEF is based on the eclipse platform. This platform itself already provides a framework for text editors. This eclipse text editor framework provides all the user interface (UI) elements that a language-dependent text editor needs. Eclipse allows developers to create there own language-dependent text editors as eclipse plugins. TEF simply extends this eclipse text editor framework with a general implementation for background parsing. Each generated TEF editor is an eclipse plugin that instantiates an eclipse language-dependent text editor and realises the editor behaviour with TEF's background parsing implementation. The background parsing of each generated editor is parametrised with a notation description that controls the language-dependent behaviour of background parsing.

TEF allows to create different kinds of textual editors from the same language description (refer to Fig. 3.20): these are language-dependent text editors, textual model editors, and editors embedded into graphical or tree-based editors. All different editor kinds are realised using TEF's background parsing implementation and operate on the same notation description. The only difference between these editors is how they load and store the edited artefact. Text editors simply load and store the textual representation as a string of characters. Textual model editors load and store models: they de-serialise and serialise models from and to XMI strings stored in the corresponding file. A textual representation is created from the loaded model, and when stored, the latest model created during background parsing is stored. Textual model editors embedded in graphical editors copy a sub-model from the graphical host editor, create a textual representation for that sub-model and allow users to change this representation. When the user saves the model, the editor copies the latest model back into the host editor's model.

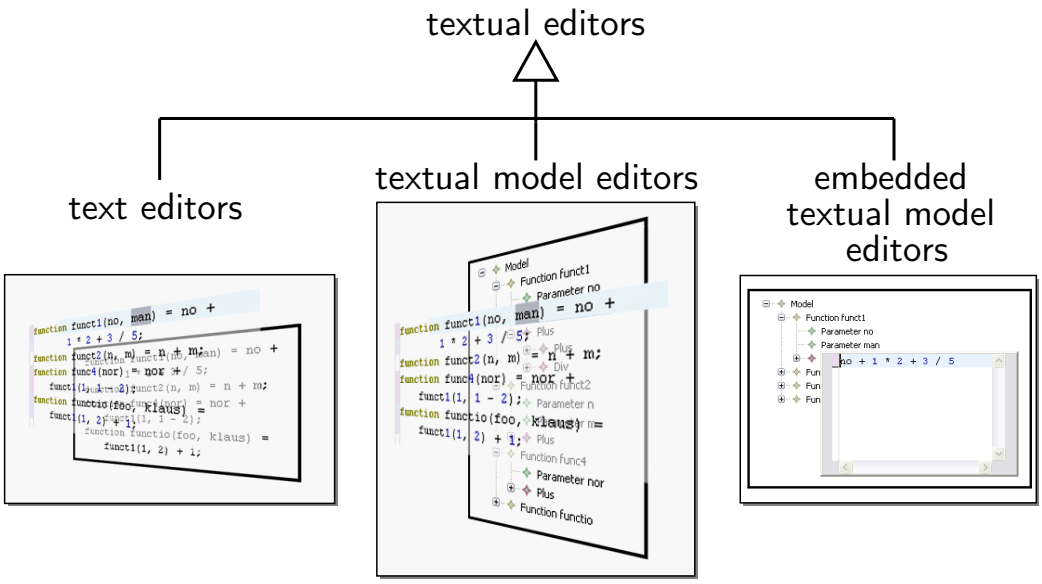


Figure 3.20: The tree different kinds of editors that can be generated with TEF.

3.7.4 Content Assist

A convenient feature of modern language-dependent text editors or textual model editors is the editor capability to provide possible continuations for the text under the cursor. This ability is called *content assist* or *code completion*. Content assist has a long history, starting from simple language-independent approaches such as *hippo completion*. Hippo completion proposes any word in a text document regardless of the syntactical context the cursor is positioned in. State of the art content assist is based on language specific notation and especially identification. This kind of content assist was for the first time introduced with the *intelliJ* Java IDE.

Today, editor users expect editors to assist in the completion of identifiers and keywords. In this section, I introduce techniques to describe content assist. These techniques are integrated into TEF, and I present a few case-studies based on TEF that were conducted to proof the general applicability of the presented content assist techniques.

You will see that it is possible to describe content assists for keywords and references (identifier) and that editors with content assist can be created from these descriptions automatically. Beyond that, it is possible to even generate the content assist descriptions, for a lower quality of content assist, solely based on notation descriptions, which have to be written anyway. Only for higher quality content assist, manual implementation is necessary

example grammar excerpt from a simplyfied OCL grammar:

```
collection_op_call = expr "->" IDENTIFIER "(" variables "|" expr ")"
op_call = expr "." IDENTIFIER "(" arguments ")"
variable_access = IDENTIFIER
expr = collection_op_call | op_call | variable_access
```

syntactical context for a symbol content assist: IDENTIFIER

three syntactical contexts for single reduction assists:

```
                                        symbol          suffix
1. collection_op_call = expr "->" IDENTIFIER "(" variables "|" expr )"
   └──────────────────────────────────────────────────────────────────┘
                                    rule
2. op_call = expr "." IDENTIFIER "(" arguments ")"
3. variable_access = expr "." IDENTIFIER
```

syntactical context for a multiple reduction assist:

```
[ collection_op_call = expr "->" IDENTIFIER "(" variables "|" expr ")",
  expr = variable_access,
  variable_access = IDENTIFIER ]
```

Figure 3.21: Some examples for syntactical contexts.

to constrain content assist proposals to those allowed by language specific static semantics.

Content Assist Types

I introduce the notion of a *content assist type*. The developer, who develops the textual model editor, can define multiple content assist types. The editor uses these content assist types to offer content assist. A content assist type defines a *syntactical context* and instructions to collect *content assist proposals*. When the user of the editor requests content assist, the editor determines for each content assist type if the current cursor position is placed in the syntactical context that is defined in the content assist type. The editor selects a set of *active* content assist types. Content assist proposals are collected for all active content assist types based on the instructions given in these content assist types. The current model, text, and cursor position is used as input for these instructions. All the collected proposals are finally presented to the user.

Determining the Syntactical Context

The following is known to the editor developer and is also programmed into the editor: the language's meta-model, the notation's grammar, and the

relation between grammar and meta-model. Based on this information the editor developer has to define possible syntactical contexts. The editor also knows the current text, a model based on a possibly earlier version of the text, and the cursor position. From these information it has to determine if a syntactical context is active or not.

What is a syntactical context? A syntactical context describes a specific point within a language construct. A syntactical context allows to specify certain positions within a text. These positions describe the wanted points within instances of the language construct. I say that these positions are *located* in this syntactical context.

How can you define a syntactical context? I distinguish between syntactical contexts of three complexity levels. The simplest syntactical context is defined by a single terminal or non-terminal symbol. An assist using such a context is called *symbol content assist*. A more complex syntactical context is defined by a symbol, used within a specific grammar rule. An assist using such a context is called *single reduction content assist*. *Multiple reduction content assists* use a symbol within a specific grammar rule, which again is used within a specific grammar rule, and so on. Syntactical contexts can be defined using the data structures in the top of Fig. 3.22; Fig. 3.21 furthermore shows four examples for syntactical contexts.

How can you determine if a syntactical context is active or not? I use LR-syntax analysis, which has two properties that are important. Firstly, once a symbol is shifted onto the parse stack there cannot be a syntax error in front of that symbol. Secondly, reductions only happen on top of the parse stack. I use the following algorithm: the document is LALR(1)-parsed up to the cursor position. Now, you emulate continued parsing as if the text following the cursor position is written according to the syntactical context. If that is possible, the context is active. For symbol content assists, this means you try to shift the symbol onto the parse stack. If that is possible, the context is active. For a single reduction content assist, you shift the symbol, then shift the symbols in the rule suffix, and then try to reduce with the context's grammar rule. If this is all possible, the context is active. For a multi-reduction content assist, you do as in a single reduction content assist, but after reduction, you try to shift the suffix of the next rule, reduce with this rule, then continue with the next rule, etc. If you can do so for all parts of the multi-reduction content assist, it is active. Fig. 3.22 shows pseudo-code for this algorithm.

definitions

```
datatype Symbol
array Symbol[] Rule
struct SymbolCA { symbol: Symbol }
struct SingleReductionCA { rule: Rule, symbol: Symbol, suffix : array Symbol[] }
array MultipleReductionCA SingleReductionCA[0..n]

boolean shift(Symbol)  // shifts the symbol on the parse stack if possible
boolean reduce(Rule)   // reduces the parse stack using the given rule if there is
                       // a follow up symbol allowing the reduction
boolean reduce(Rule, Symbol) // reduces the parse stack using the given rule if
                             // possible using the given follow up symbol
```

algorithm that takes a content assist *ca* as input

```
parse the document using the notation's grammar rules and LR-syntax analysis
stop parsing at the current cursor position
switch type of ca
    SymbolCA:
        return shift(ca.symbol)
    SingleReductionCA:
        if not shift(ca.symbol) then return false
        for symbol in ca.sufix do
            if not shift(symbol) return false
        if reduce(ca.rule) return true
    MultiReductionCA:
        for i = 0; i < ca.length; i++ do
            if not shift(ca[i].symbol) return false
            for symbol in ca[i].sufix do
                if not shift(symbol) return false
            if (i+1 < ca.length)
                if not reduce(ca[i].rule, ca[i+1].symbol) return false
            else
                if not reduce(ca[i].rule) return false
        return true;
```

Figure 3.22: An algorithm that determines if an input content assist (ca) is active in pseudo-code.

Collecting Content Assist Proposals

Each content assist type includes instructions to collect proposals. The proposals can be collected from the following information: the validity of the syntactical context of the content assist, the parse tree created during analysing the syntactical context, and the current model. I distinguish between three different *quality levels* for proposals.

For the lowest proposal quality, you only use the syntactical context without any additional instructions. From the syntactical context, you know which type of element can be inserted at the cursor position and you simply offer all instances of the corresponding type. Take the OCL example in Fig. 3.21 and the single reduction assist number 2: you propose all identifiers that reference an operation. Therefore, you collect all operations in the given model. The problem is that all the proposals are valid based on the notation's grammar, but not necessarily based on its identification mechanism. In the example, you are not constraining the set of operations based on the expression type that the operation is called upon.

For a higher quality level, you again use the syntactical context, but now also allow additional constraints based on the parse tree. Starting at the node, located in the syntactical context, one can visit all the *containers* of the syntactical context by navigating the parse tree towards its root. Here, container refers to containment as defined by composition in the meta-model. Take the multi-reduction content assist in Fig. 3.21: navigating from the variable access to the collection operation call (two containment relations out and therefore two parse tree nodes up), you can access the variables of the collection operation call, and these are the variables you want to propose.

For the highest proposal quality, you allow proposal constraints based on parse tree and model. Previously, using only the syntactical context, you proposed all operations in the first single reduction content assists of the OCL example in Fig. 3.21. But based on parse tree and model, you can determine the expression that the operation is called on. You can determine the expression's type and all operations allowed for this type. It is now possible to constrain the set of operations to those allowed by OCL's operation call semantics.

Realisation and Case-studies

The Textual Editing Framework (TEF) and content assist TEF editors provide basic content assist based on a textual notation description only. The automatically generated content assist comprises *keyword assists* and *identifier assists*. Keyword assists are generated for each keyword or

special character in the notation. They use a symbol content assist type to define a syntactical context, and their proposal collection always provides the terminal itself as the only proposal. Identifier assists are generated for each reference binding in the textual notation description that refers to a meta-model reference that defines a non-containment reference. These identifier assists use a single reduction content assist type to define a syntactical context, and their proposal collection provides a list of names from all those model elements that have the meta-type of the corresponding reference.

There are two ways to customize the automatically generated content assist types. Firstly, a language developer can implement callbacks that alter the proposal collection behaviour of the automatically generated identifier assists. Giving parse tree and model as input, these methods can be used to realise the more advanced proposal quality levels. Secondly, a language developer can additionally define own content assist types. Here, the developer has to implement certain interfaces to define a syntactical context using Java data-structures similar to those in Fig. 3.22 and he needs to program a proposal collection. Model editors for several example toy languages, like the expression language in Fig. 3.13, and more significant editors for EMF's meta-modelling language Ecore [117] models and OCL [76] constraints were built. These case-studies showed that high quality content assist for such languages can be created, while manual syntactical contexts have only to be provided in some seldom cases, and manual proposal collection instructions are only necessary to cover aspects of language specific static semantics.

Content assist for OCL The TEF generated OCL editor, for example, automatically provides a content assist for each operator. Since these keyword assist are defined for a certain syntactical context, operators are only proposed if this operator is syntactically allowed at the corresponding position. However, these automatically generated assists only reflect OCL's syntax, and operators are proposed even if they are not allowed semantically, e.g. not allowed based on operand types.

OCL contains five kinds of references: references to local variables, i.e. *self*, operation parameters, and variables defined in *let*-statements; references to the properties of model elements; references to the operations of model elements; references to types; and references to operations of OCL's collection library. Content assists for all references could be generated. However, the proposal collection had to be manually altered. Most references do not reference elements within the OCL model, but within the model that the OCL is written for. And, you only want to allow proposals that are semantically valid, e.g. only propose properties and operations defined in the

corresponding type. The first alteration was necessary, because TEF's automatically generated identifier assists only rely on the edited model, and not on some external model. The second alteration is always necessary if the desired content assists have to reflect the language's static semantics.

Creating the OCL editor leads to the general conclusion that even though TEF provides reasonable syntactical contexts automatically, and therefore relieves the language developer from navigating through abstract syntax trees, etc., there is still lots of manual programming necessary if content assist has to be restricted to semantically reasonable proposals.

3.7.5 Embedding Textual Model Editing into Graphical Model Editing

In the past, the superiority of (purely) graphical representations was widely assumed at first and often challenged [40, 84] later. Moher et al., for example, concluded in [65]: *"Not only is no single representation best for all kinds of programs, no single representation is [...] even best for all tasks involving the same program."* Today, graphical modelling languages and domain specific languages (DSLs) often use a mixed form of representation: they use diagrams to represent structures visually, while other elements are represented textually. Examples for textual elements are signatures in UML class diagrams, mathematical expressions in many DSLs, OCL expressions used in other modelling languages, or the many programming constructs of SDL [46].

Existing graphical editors address textual model parts poorly. The OCL editors of many UML tools, for example, barely provide syntactical checks and keyword highlighting. As a result, modellers produce errors in OCL constraints: errors that stay unnoticed until later processing; errors that when finally noticed, are hard to relate to the OCL constraint parts that caused them. For other constructs, like operation signatures in UML class diagrams, editors often provide no textual editing capabilities at all. So editor users *click* signatures together. This process is slower and less intuitive than writing the signature down. As a general conclusion, editing the textual parts of models is less efficient than existing text editor technology allows.

In this section, I look at techniques to combine textual modelling with graphical modelling, to embed textual model editors into graphical editors[10].

[10]The applicability of the presented techniques is not limited to graphical editors, but to editors based on the *Model View Controller (MVC)* pattern [8]. This also includes tree-based model editors, which are very popular in EMF-based projects.

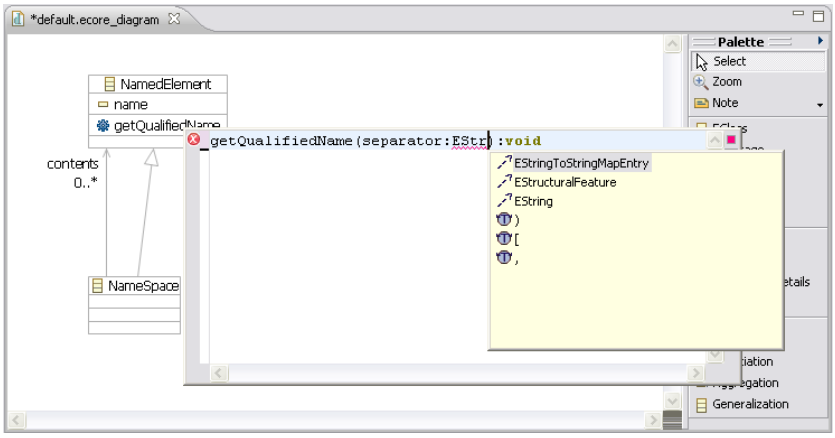


Figure 3.23: The Ecore GMF editor with an embedded textual model editor.

Definition 35 (embedded textual modelling) *With* Embedded textual modelling *we denote the process of writing and manipulating models with an editor capable of graphical editing (or a different form of Model View Controller (MVC)-based editing) and textual editing. Such editors use an* embedded text editor *to edit textual parts of a model representation. The MVC-based editor is called* host editor *since it is the primary editor and handles the model represented in both the host and embedded editors.*

Editor users open an *embedded text editor* by clicking on an element within the graphical *host editor*. The *embedded editor* is shown in a small overlay window, positioned at the selected element (see Fig. 3.23). Embedded editors have all the editing capabilities known from modern programming environments. To implement this approach, TEF was used to create embedded textual editors for graphical editors developed with the Eclipse Graphical Modelling Framework (GMF) [119]. TEF allows to create textual notations for arbitrary EMF meta-models. This includes the meta-models used for GMF editors.

As potential impact, this work can enhance the effectiveness of graphical modelling with languages that rely in parts on textual representations. Furthermore, it encourages the use of domain specific modelling, since it allows the efficient development of DSL tools that combine the graphical and textual modelling paradigm. In general, this work could provide the tooling for a new breed of languages that combine the visualisation and editing advantages of both graphical notations and programming languages.

You will see how embedded editors work in general and what problems have to be solved to realise embedded textual modelling. After this general discussion, I briefly discuss how embedded textual modelling is realised in

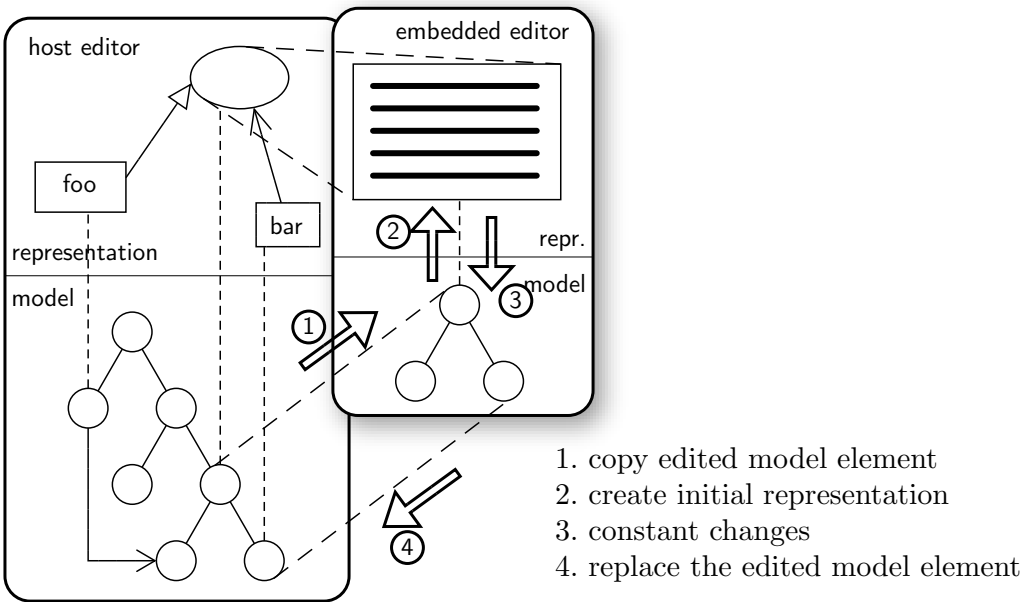


Figure 3.24: Steps involved in the embedded textual editing process.

TEF, and look at a few example applications, which prove that embedded textual modelling can be generated from notation descriptions in the same way than other language-dependent text and textual model editors.

An Embedded Textual Modelling Approach

Graphical model editors are based on the *Model View Controller* MVC pattern [8]. An MVC editor displays representations for model elements (model) through view objects (view). It offers actions, which allow the user to change model elements directly. Actions are realised in controller objects (controller). Examples for such actions are creating a new model element, modify the value set of a model element's feature, deleting a model element. The representing view objects react to these model changes and always show a representation of the current model. In MVC editors the user does not change the representation, only the model; the representation is just updated to the changed model. From now on, I assume that the host editor is an MVC editor.

The embedded editing process is illustrated in Fig. 3.24. The user selects a model element in the host editor and requests the embedded editor. The embedded editor is opened for the selected model element (1). I call this model element the *edited model element*. The edited model element includes the selected model element itself and all its components. The opened

textual model editor creates an initial textual representation for the edited model element (2). The user can now change this representation, and background parsing creates new partial models, i.e. creates new edited model elements (3). The model in the host editor is not changed, until the user commits changes and closes the embedded textual model editor. At this point, the embedded editor replaces the original edited model element in the host editor's model, with the new edited model element created in the last background parsing iteration (4).

There are three problems. Firstly, you need textual model editors for partial models. Obviously, it is necessary to describe partial notations for corresponding partial representations. Secondly, when the editor is opened, it needs to create an initial textual representation for the selected model part. Finally, when the editor is closed, a new partial model was created during background parsing. This newly created partial model needs to replace the original edited element. All references and other information associated with the original edited model element have to be preserved.

Creating Partial Notation Descriptions

Textual model editors rely on textual notations. Whether these notations cover a language's complete meta-model or just parts of it, is irrelevant, as long as the edited models only instantiate those meta-model parts that are covered by the textual notation.

Two solutions are possible: firstly, language developers only provide a partial notation for the meta-model elements that they intend to provide embedded textual modelling for. In this case, the developers have to be sure that they cover all related meta-model elements. Textual editing frameworks can automatically validate this. Secondly, language developer provide a complete notation, and the editing framework automatically extracts partial notations for each meta-model element that embedded editing is intended for.

Initial Textual Representations

To create the initial textual representation, you can use pretty printing as explained in section 3.7.2. The automatic generation of white-spaces is the a reasonable choice for embedded textual editing. The edited text usually only comprises text pieces; white-spaces with hidden information, like empty lines, are not that important. Furthermore, the embedded text editors rely on the modelling facilities of the host editor; storing information beyond the model requires to change the host editor's implementation. And finally, with

automatic layout, it is also possible to textually represent models that were not created via a textual representation. Models created with other means than a textual model editor (e.g. the host editor) can also be edited within such an editor. To realise automatic generation of white-spaces, you can use the already presented techniques.

Committing Model Changes

Problems caused by different editing paradigms The host editor changes the model with small actions that only effect single or very few model elements. Opposite to the MVC host editor, the embedded textual model editor, based on background parsing, creates complete new model elements for each representation change. This causes two problems. Firstly, other model elements, not part of the edited model element, might reference the edited model element, or parts of it. These references break, when the original edited model element is replaced by a new one. Secondly, the edited model element might contain information that is not represented in its textual representation; this information will be lost, since it is not part of the model element created through background parsing.

In today's modelling frameworks, you know all the references into an edited element and its parts, and you can reassign these references to the replacement model element and its parts. This would solve the first problem. You could also merge changes manifested in the newly created model element into the original model element. This would only update the original edited model element and not replace it. This would solve both problems. Anyway, both solutions require identification: the editor has to access whether a original model element is meant to be the same as an corresponding model element of the newly created model element. The editor can achieve this based on the elements' identity. This is obviously language specific, and identification has to be defined for each language.

With identification, the editor can tell whether two model elements have the same identity, and realising the first problem solution becomes very easy. The editor takes all references into the original edited model element, determines the identity of the referenced model elements within the original editor model element, searches for a model element with the same identity within the newly created model element, and reassigns the reference. The second problem solution requires some sort of algorithm that navigates both, the edited model element and the newly created model element, simultaneously along the model elements' composition. The merge algorithm has to compare the model elements feature by feature based on their identity, and transcribes all differences into the original edited model element. Deeper dis-

cussions about model merging is outside of this thesis' scope; model merging algorithms and techniques are described in [126, 57].

One problem remains: When the user changes the text representation in a way that the identity of an element changes. It is not clear what the users intentions are. Did the user want to change the name of a model element or replace a model element with a new element. The editor can only assume that the user wanted. One way to solve this problems is to give the user the possibility to express his intention, e.g. provide a refactoring mechanism that allows to explicitly rename a model element. Another solution used by the IPSEN project [52] involves marking character-wise changes and reflect those back on model elements via a reverse notation.

Problems caused by different undo/redo paradigms A convenient feature of model editors is the possibility to undo and redo model changes. This needs to be preserved for model changes in embedded text editors. In MVC editors, model changes are encapsulated in command objects, which allow to execute single model changes, and to reverse the execution of single model changes. Commands for executed model changes are stored in a command stack, which the editor uses for undo/redo. This is different in a textual model editor based on background parsing, where users change strings of characters. User actions are represented as replacements on that string. Undo/redo is based on a stack of string replacements. Embedded textual model editors and their graphical host editors obviously use incompatible representations for model changes.

I propose the following solution. Embedded text editors offer string replacement based undo/redo during textual editing. When the user closes the embedded editor and commits the textual changes, the necessary actions to replace the original edited model element are encapsulated into a single command, which is then stacked into the host editors undo/redo facility. This is a compromise: it allows to undo whole textual editing scenes, but does not allow to undo all the intermediate textual editing steps once the embedded editor is closed.

Realisation and Experiences

A framework for embedded textual model editors TEF was extended for the development of embedded editors. Embedded editors can be created for EMF generated tree-based editors and editors created with the Graphical Modelling Framework (GMF) [119]. These embedded editors do not require to change the host editor. In theory, TEF should work for all EMF-based MVC host editors. To use TEF for embedded editors, language

developers provide a notation description for those meta-model elements that embedded textual editing is desired for. TEF creates the embedded editors and provides so called object-contributions for corresponding EMF objects. These object-contributions manifest as context menu items in the host editor. With these menu items, users can open an embedded text editor for the selected model element. The embedded editor is a full fledged TEF editor providing all its features, except for the outline view, which is still showing the host editors outline. The embedded editor will only show the textual representation of the edited model element. The embedded text editor can be closed in two ways. One way indicates cancellation (by clicking somewhere into the host editor); the other way commits the changes made (pressing shift-enter).

TEF uses the following problem solutions. TEF creates partial notation descriptions by reducing a given notation for the specific edited model element dynamically, when the editor is opened. TEF editors create initial representations based on pretty printing with automatic white-space generation using layout managers. Embedded editors commit model changes by creating one single compound command that is added to the host editors command stack to preserve the host editors undo/redo capability. This command contains sub-commands that replace the original edited element and reassign all broken references based on either TEF's default identification or a language developer given language specific identification mechanism.

Textual editing of Ecore models TEF was used to develop a textual notation for Ecore and to create embedded textual editing for the graphical Ecore GMF editor and the standard tree-based Ecore editor. This allows a more convenient editing of signatures for attributes, references, and operations. With the textual editing capabilities this becomes indeed more convenient and renders the process of, e.g., creating an operation with many parameters more efficient.

The work on the Ecore editors affirmed a few of the mentioned problems. Firstly, many elements in the Ecore language carry information drawn from many side aspects of meta-modelling, such as parameters for code generation or XMI generation. Including all this information in the textual notation, would render it very cumbersome. Omitting this information in the textual notation, however, causes the loss of this information, when the corresponding model parts are edit textually. The hope is to eliminate this problem by applying a model merging approach to update the original edited model element instead of replacing it. Secondly, a textual notation for Ecore needs complex identification constructs to realise textual references. Identification

in Ecore has to rely on namespaces, imports, local and fully qualified names. Non of those constructs were defined in the language itself, and had to be invented on top of the actual Ecore language. The automatic created editing facilities had to be extended with manual implementations that describe the needed identification constructs.

An OCL editor integrated into other model editors The OCL constraint language is often used in conjunction with languages for object oriented structures, or the behaviour of such structures. Hence, OCL expressions are often attached to the graphical notations of languages like UML, MOF, or Ecore. Therefore, editors for those languages should support OCL editing, but they usually only do by means of basic string-based text editing.

TEF was used to develop an OCL editor based on the MDT OCL project. As an example, I integrated this editor with the tree-based Ecore editor: The EMF validation framework requires OCL constraints stored in Ecore annotations, because Ecore itself does not support the storage of OCL constraints. Since this only allows to store OCL constraints in their textual representation as strings, the embedded textual OCL editor is actually a normal text editor, which only uses background parsing to create internal OCL models to support advanced editor features, i.e. code completion and error annotations. This makes committing the changes to OCL constraints particularly easy, since the embedded editor only has to replace a string annotation in the host-editors Ecore model. Another example application that I integrated the OCL editor into is the graphical editor for the UML activity-based action language in [108, 111].

Editing for mathematical expressions in DSLs Many of today's DSLs are developed based on EMF and instances of these languages are consequently edited using EMFs default generated tree-based model editors. This is fine for most parts of these languages, but can become tiresome, if models contain mathematical expressions. Since, mathematics is the most common mean to express computation, such expressions are part of many languages.

TEF was used to developed a simple straight forward notation for a simple straight forward expression meta-model. This expression meta-model and notation is a blueprint for integrating sophisticated editing capabilities for expressions into DSLs. This can be used to realise expressions in a domain specific language for the description of cellular automatons, used to predict the spread of natural disasters like floods or fire.

3.7.6 Alternative Textual Model Editing Approaches

Work on textual notations based on meta-modelling with MOF includes Alanen et al. [2], Scheidgen et al. [31], Wimmer et al. [125]. This basic research was later utilised in frameworks for textual model editors. These are either based on existing meta-models (TCS [48], TCSSL [69], MontiCore [59]), or they generate meta-models or other parse tree representations generated from the notations (xText [83], Safari [13, 45]). All these frameworks and projects facilitate background parsing as editing strategy. Background parsing, its notation meta-languages, and used algorithms are inspired by attributed grammars as described in [55].

All these background parsing-based frameworks provide similar notation descriptions and generated editor features. However, they differ in the way grammar elements are mapped to meta-model elements. Some frameworks, like xText, do not map grammars to existing meta-models, but create a meta-model-based on the notation. XText thereby allows to describe notation and language in once. Other frameworks, like TCS, separate notation and meta-model, as TEF does, but have two different description languages for grammar, and grammar to meta-model mapping. Other frameworks, like safari, don't use any MOF-like meta-modelling at all, and use parse trees as models. The frameworks use different approaches to references and identification. All frameworks that support identifier resolution, provide a simple default mechanism and only allow customised identification based on programming APIs. Depending on the identifier resolution capabilities only xText and safari support reasonable content assist.

Safari allows manual implementation of content assist. This means that the language developer has to realises each content assist type as a piece of Java code, which takes an AST of the document and the current cursor position as input and produces a collection of proposals as output. It provides no automatic means for determining if a syntactical context is active or proposal collection.

The xText framework provides automatic reference assists similar to those of TEF, and allows to manually implement content assist. To manually implement content assists the editor developer has to provide pieces of Java code that take a *grammar-element* and the model that the edited text represents as input and provide a collection of proposals as output. The given grammar-element represents the point in the grammar that reflects the current cursor position. This information allows to realise symbol and single reduction content assists easily. However, there is no support to realise multiple reduction content assists.

The background parsing frameworks, only support the editing of text files.

Models have to be created from those text files separately. Furthermore, pretty printing capabilities, if supported, are not directly integrated into the editors (since they are only text editors). This makes it hard to facilitate those frameworks for embedded textual editing, because these editors can not create an initial representation.

Besides using context-free grammar and background parsing, textual modelling can also be conducted using the MVC pattern. MVC is used to realise Intentional Programming [110] and the Meta Programming System (MPS) [25].

Such MVC editors don't allow users to type arbitrary text, but to use predefined commands to insert language construct instances. Content assist plays an integral role in these editors, since models are edited by selecting commands from a list determined by the current syntactical context. But content assist also works intrinsically different, because the text is not edited by the user, but created by the editor.

Since the editing paradigm is the same than in graphical editors, it is thinkable that MVC-based editing frameworks can be integrated into graphical editors as well, maybe with more natural solutions to most of the presented problems. However, using MVC, only allows to create models based on simple actions. This way, you have to create, e.g., an expression as if you would create its parse tree: from top to bottom. This is the same kind of limitation that is already imposed upon graphical editors, and this is exactly why you want to use textual modelling in the first place.

The GMF framework itself, provides some very simple means to describe structured text. It allows to create simple templates that assign different portions of a text to different object features. These simple templates allow less than regular complexity, and are therefore inadequate for many textual language constructs. First premature steps to describe the relations between graphical notations and textual notations have been made by Tveit et al. [87].

3.8 Conclusions

In the beginning of this chapter, I listed a series of problems with textual notations: (1) textual model editors are expensive, (2) meta-models are more powerful than grammars, (3) textual representations are often partial representations, (4) high quality content assist, (5) non-unique and ambiguous notations.

To address the first problem (1), I presented techniques and an implementing framework for the efficient DSM of textual model editors. This makes the development of sophisticated editors economical even for the small-

est domain specific languages. In bridging the expressiveness gap between meta-models and grammars, the issue of resolving references within textual representation still provides a big challenge (2). TEF and all other frameworks for textual model editing only provide primitive mechanisms to address this issue. The underlying problem are the numerous possible identity concepts. It is hard to find a reasonable amount of abstractions that can be used to describe all possible identity concepts. As a result, it is hard to find one notation description language that can be used to describe all the specific identity constructs of all the specific languages. Conclusively, identity concepts have to be realised with multi-purpose programming languages. I provided the means to create embedded textual model editors. This allows to create editors for languages with notations that are only partially textually represented (3). I elaborated on creating high quality content assist as automatic as possible (4). Yet, good content assist depends on the underlying identity concepts, and can only be implemented by means of manual implementation in a multi-purpose programming language rather than abstract description in a notation description language. Non-unique and ambiguous notations (5) were barely touched in this thesis and even taking the existing related work into account, these problems remain unaddressed. Non-unique notations mostly refer to notations containing additional information, not covered in the represented model. Examples are white-spaces and commentaries. This is rather a small technical issue, and the easiest solution is to provide a hybrid between text and model editor, which simply stores both artefacts: the representation and the model. Ambiguous notations, however, provide the bigger challenge. This problem refers to notations that only partially cover the model. I addressed this from a specific view point when I introduced a way to embed textual modelling into graphical modelling. Here, the problem is again caused by insufficient understanding and handling of model element identities. Background parsing creates a new model with each editing step. If you use this strategy to edit partial models, you always have to merge the newly created partial model into the actual model or otherwise loose the not covered model information. This merging process is only possible when the identity of model elements is understood. This problem not only occurs in embedded textual and graphical modelling, where partial model means a sub-model, but also in textual notations that cover only certain information about a model. This problem becomes especially critical when one thinks about multi-user, multi-view editing environments, where a single model is edited by possibly many user from different views. Each view covering partial model information, and each view uses a different notation. Merging all the different editing results is heavily based on the identity of the involved model elements.

Despite all the existing problems, DSM of textual editors has become a very popular method in the development of domain specific languages, and is even considered for created editors for programming languages (e.g. eclipse JDT and safari [13]). Even though all the existing frameworks are intrinsically the same (background parsing and unsolved identity issues), they all feature some specialities that distinguish them from each other. Hopefully, there will be a DSM framework for textual modelling that unifies the capabilities of all the existing meta-languages and meta-tools.

Beyond this thesis, my framework TEF is used to create editors for DSLs within the METRIK graduate school. This includes a DSL for the modelling of fire and water propagation and a textual editor for Petri-Nets. Furthermore, TEF is used to realise an IDE for Python.

At different places in this chapter, I came across the problematic description of identity concepts. This is probably the most important future research issue. On approach to tackle this problem would be to analyse existing languages and textual notations and attempt a classification of the used identify concepts. Based on this, one could try to build a set of abstractions that can be used to describe all existing (and probably all future) identity concepts. However, it would't be a surprise, if this work would conclude that existing identity concepts are to different to be described by other means than a very broad multi-purpose language. Beyond programming languages, this could also mean that identity can be adequately described by query languages such as OCL or by model transformations written in languages such as QVT. Another possibility to address this problem could be the integration of parsing and identity resolution based on concepts researched in the field of attribute grammars. One example is used in the *EPromote* project [97], where the authors use Prolog to mimic attribute grammars with model transformations between a input stream of character and an output model.

Chapter 4

Semantics – Describing Operational Semantics

In previous chapters, I addressed the description of languages as a set of instances, and I discussed how these instances can be represented with notations for languages. Thus, the aspects syntax and representation were dealt with. This leaves language semantics for this chapter. Computer languages are used for many different domains and therefore their semantics can use completely different kinds of semantic domains. In this thesis, the focus lays on executable languages, i.e. language with instances that have a running system as meaning. When you want to describe languages that are used to create executable systems, you need to describe how the distinct constructs of a language form a running system: you need to define an operational semantics for these languages. After a general introduction into this subject, this chapter focuses on the description of operational semantics based on MOF-like meta-modelling.

4.1 Introduction

Previously, I simply stated that semantics consists of a semantic domain and a mapping between language and this semantic domain. I did not further characterised any properties for the semantic domain. Hence, the term semantics allows to associate all kinds of meanings to language instances, but I want to focus on operational semantics in this thesis. This means I am interested in language semantics that assigns each language instance a behaviour. The behaviour of a running system are a series of computational steps taken by the system. Such a series of steps can be mathematically handled as a half-order that defines causality between distinct computational steps.

There are several ways to look at a computational step. Firstly, you can say each computational step is accompanied by an event within the running system. And secondly, you can say with each computational step the running system transits from one state into another. I will later follow an approach where I describe operational semantics in way that allows to execute a language instance (described system) based on a semantics description. This does not focus on what a system does in terms of events, but on how a system works in terms of progressions of system states. Therefore, operational semantics is defined for this purpose:

Definition 36 (operational semantics) Operational semantics *is semantics with a semantic domain that consists of behaviours. A* behaviour *is a half order describing a progression of* system states *or representations for system states. The half order defines a causality between states.*

To describe operational semantics, you need to describe the semantic domain and the semantic mapping depending on an already described language. The description of a semantic domain of some operational semantics can be further divided into a description of possible system states and transitions between those states, because combinations of system states and transitions between them constitute the members of such a semantic domain.

Whatever states or state representations are used in a semantic domain, you need formalisms to define the set of all possible states. This set of possible states might be one and the same for all language instances in a language, or it might depend on a language instance. In a description for the semantics for a programming language for example, the representation of heap space does not depend on a specific program, but the representation of variables and variable assignments does depend on the concrete variable definitions in a concrete program. *State description formalisms* allow you to describe possible infinite sets of states with finite *state descriptions*.

A behaviour is either a set of sequences of states (also a half-order), therefore called *sequential behaviour*, or a half-order of states, called *distributed behaviour*. Distributed behaviours allow to describe the dynamics of language instances with more than one possible thread of computational steps. Based on the given operational semantics and behaviour, a formalism that can be used to describe such operational semantics can be defined as follows:

Definition 37 (semantics description formalism) *The term* semantics description formalism *denotes a formalism that allows to describe a formalism $F = (L, SM, SD)$ for a given language L. An* operational semantics description formalisms *is a semantics description formalisms where the semantic domain SD for the described formalisms consists of behaviours or sets of behaviours.*

I want to facilitate operational semantics formalisms in DSM for interpreters. The means a meta-language to describe the operational semantics of a language and a meta-tool for this meta-language is needed. Meta-language semantics and meta-tool behaviour is thereby defined by the used operational semantics formalism. An corresponding meta-tool can either generate interpreters from a semantics description or it interprets language instances directly based on a semantics description. Either way, interpretation means that the system described by a language instance is executed based on the language instance and the operational semantics description.

4.1.1 Structural Operational Semantics

Plotkin's structural operational semantics [85] is an often executed and well-understood approach to define the operational semantics of programming languages.

State Transition Systems

Plotkin uses *state transition systems*:

Definition 38 (state transition system) *A* state transition system *is a tuple* $(\Gamma, \rightarrow)$*, where* Γ *is a set of* states γ *and* $\rightarrow \subseteq \Gamma \times \Gamma$ *are the possible* transitions *between states. Such a transition system defines possible* behaviours *as a sequence of states. Each behaviour starts in a predefined* initial state.

The behaviour of a state transition system is defined inductively. Given the state transition system $(\Gamma, \rightarrow)$ the behaviour for an initial configuration $\gamma \in \Gamma$ is defined by:

Definition 39 (behaviour)

$$\begin{aligned} (\gamma) &\in \mathit{behaviour}(\gamma) \\ (\gamma_1, \ldots, \gamma_n, \gamma_{n+1}) &\in \mathit{behaviour}(\gamma), \mathit{if}\ (\gamma_1, \ldots, \gamma_n) \in \mathit{behaviour}(\gamma) \wedge \\ &\qquad\qquad \gamma_n \rightarrow \gamma_{n+1} \end{aligned}$$

In addition to Plotkin's state transitions systems, one can distinguish between the general behaviour generated by a state transition system, and further constrained behaviours. This allows to describe general and further constrained behaviour in different meta-languages. The definition of a state transition system alone, might only provide a weak expressiveness and makes it hard to define semantics. Instead, you can add another element to a semantics description: a function that selects specific behaviours from a general behaviour. This allows you to further constrain the behaviour defined

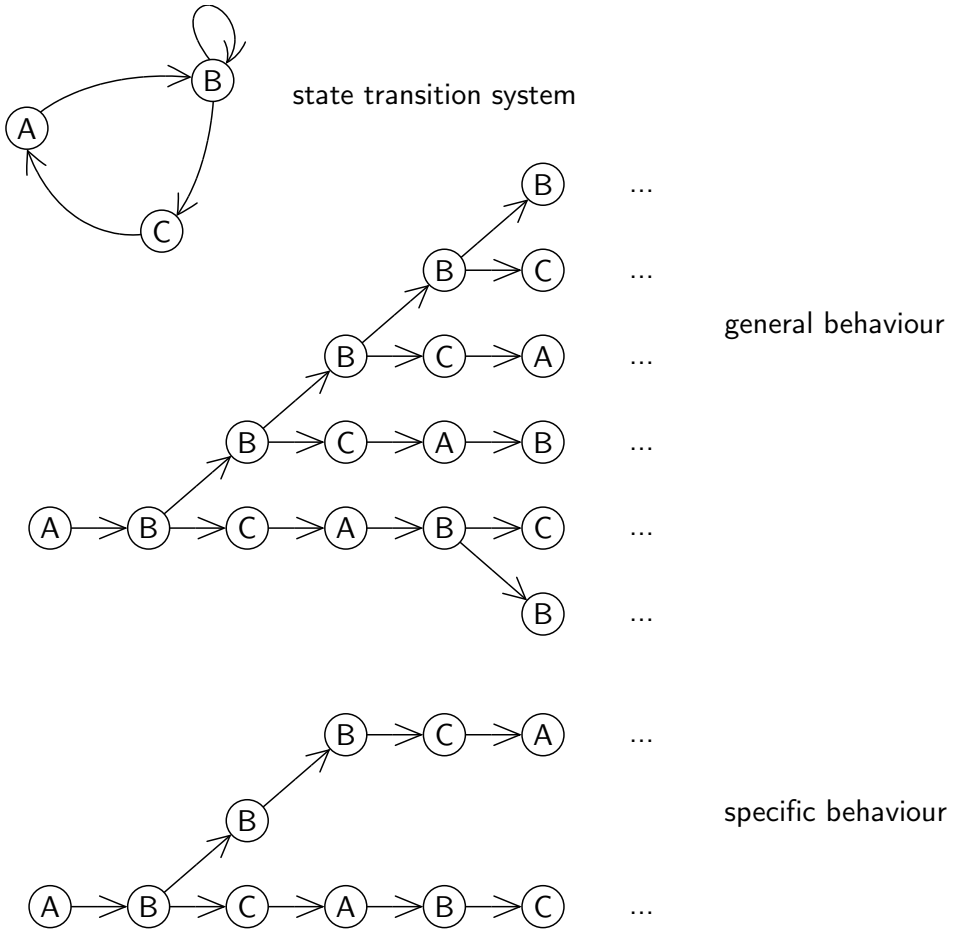


Figure 4.1: An example state transition system and its behaviour.

through a state transition system. I call $behaviour(\gamma)$ the *general behaviour* of γ, and subsets $sb \subseteq behaviour(\gamma)$ *specific behaviours*.

Definition 40 (constraint function) *A* constraint function *is a function* $c(\gamma) : behaviour(\gamma) \rightarrow \mathbb{B}$ *that selects specific behaviours from the general behaviour of a state transition system. A transition system* $(\Gamma, \rightarrow, c)$ *is called a* constrained state transition *system. The constrained behaviour is defined as* $behaviour_c(\gamma) = \{b | b \in behaviour(\gamma) \wedge c(\gamma, b)\}$.

State Transition Systems to Describe Language Semantics

The state transition system concept provides a framework that is generic enough to describe the operational semantics of many programming languages and many languages used to model software systems. I focus on Plotkin for the rest of this thesis. Plotkin can be used to structure operational semantics: each operational semantics must describe a set of possible system states, possible transitions between states, and something that determines which possible transition or transitions are actually used to progress through states. The operational semantics in an operational semantics formalism must contain descriptions of these elements.

A state transition system gives you a behaviour based on an initial state. An operational semantics describes a mapping that gives you a behaviour based on a language instances. When all language instances of a language are part of states in a state transition system, a state transition system describes an operational semantic mapping for that language. For a language L and a state transition system $(\Gamma, \rightarrow, c)$, the function $initial : L \rightarrow \wp(\Gamma)$ defines all possible initial states for a language instance in L. What distinguishes different initial states for the same language instance could be things like the environment that the language instance is executed in, the behaviour of entities interacting with the executed system, like users or outside system components, as well as input for the system.

Definition 41 (operational semantics, *math.*) *An operational semantics for a language* L *as a tuple*

$$S = (L, \Gamma, \rightarrow, c, initial)$$

Such a semantics representation defines a formalism $F = (L, SM, SD)$

$$SM(l) = \{\gamma \in initial(l) | behaviour_c(\gamma) = true\}$$

$$SD = \{l \in L | SM(l)\}$$

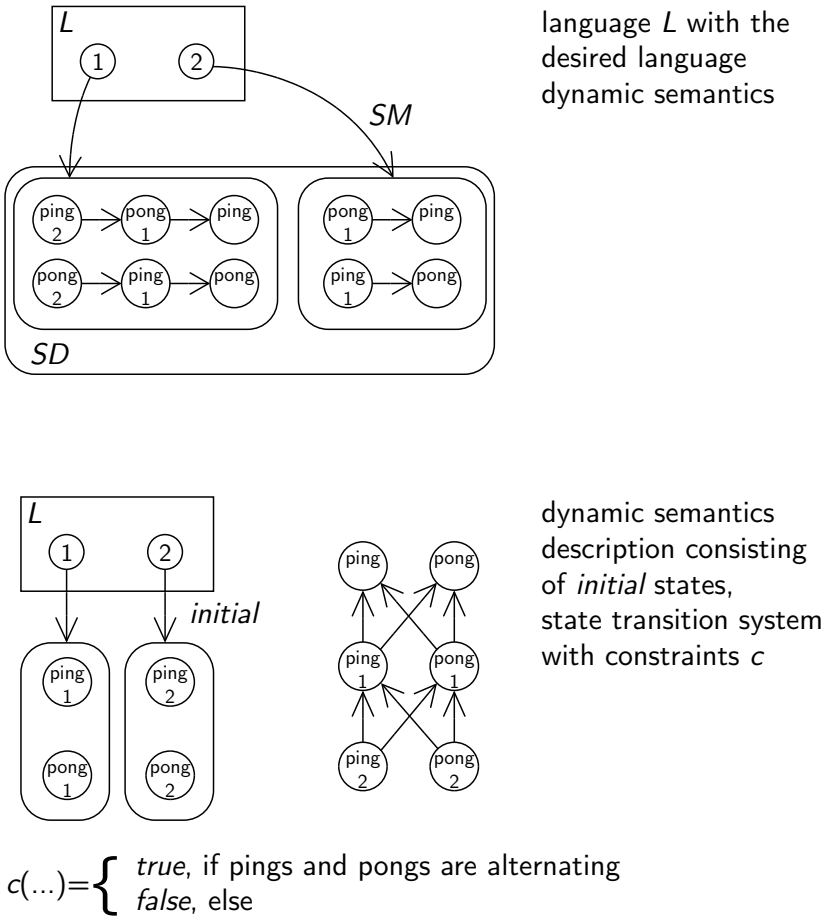


Figure 4.2: An example language and its operational semantics.

An operational semantics description formalism based on Plotkin can describe the semantics S of a language L. This means an operational semantics description formalism has to allow to describe a set of possible system states Γ based on a language L, a set of possible transitions $\rightarrow$, an optional constraint function c and an initial function *initial*.

Fig. 4.2 shows a very simple language and its semantics for demonstration purposes. The language only consist of two instances: *1* and *2*. The semantics of this languages is an alternating sequence of *pings* and *pongs*, where the length of this sequence is determined by the language instance: instance *1* has one ping-pong exchange, instance *2* two exchanges. The used state transition system generates all sequences of *pings* and *pongs* in all lengths. The *initial* function assigns the language instances to those states that are succeeded by sequences of *pings* and *pongs* of the corresponding length. The constraint function simply omits all behaviours that contain non alternating

sub sequences of *pings* and *pongs*.

You could alter the example state transition system in a way that it would only generate alternating sequences of *pings* and *pongs* (just remove the arrows between two *pings* and two *pongs*). In that case the constrain function would be unnecessary. Why is the constrain function needed? If you use a constrain function, what is described in the constrain function and what in the state transition system?

In principle a constrain function is not always necessary, but as stated earlier, the separation of state transition system and constraint function allows to describe both with separate meta-languages. The idea is that the one meta-language might have limited expressiveness. A meta-language that language developers want to use, because it is simple enough to write state transition systems easily, could be too simple to describe the state transition systems that language developers want to create. This is especially true, when you want to use existing languages to define state transition systems or constrain functions. The separation into two languages helps you to multiply expressiveness. What part of behaviours should be modelled within a state transition system and what part within a constrain function cannot be answered generally. This depends on the concrete meta-languages used, and the concrete semantics that you want to describe. [1]

4.2 Applying Plotkin to MOF-like Language Descriptions

Based on Plotkin's structure for operational semantics, I want to develop an operational semantics description formalism for MOF-like meta-modelling formalisms. I realise this formalism in a framework for operational semantics based on a MOF-like meta-modelling framework. Firstly, I discuss possible ways to describe the elements of Plotkin semantics: states Γ, transitions $\rightarrow$, constraint function c, and initial states *initial*. Secondly, I discuss problems with operational semantics and related work in describing and realising operational semantics based on meta-modelling. Thirdly, I combine descriptions for the Plotkin elements into a cohesive meta-language and briefly look into a realising meta-tool. In a concluding section, I will analyse my meta-language and meta-tool in regard to the discussed problems and related work.

[1] Another example case where the description for a single problem is separated into two are MOF-like meta-models. You use meta-modelling formalisms to describe sets of language instances, and use query languages, like OCL, to constrain the described sets of language instances.

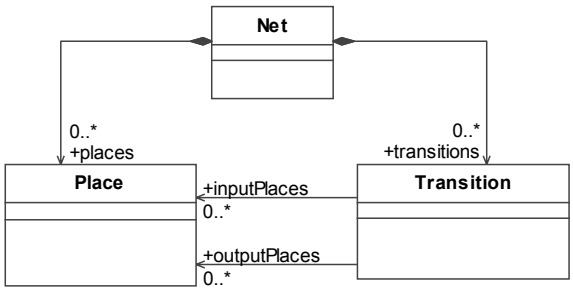


Figure 4.3: A meta-model for Petri-nets.

4.2.1 States

I simply use MOF-like meta-modelling formalisms to define sets of states. The language descriptions become descriptions for languages of states and each language instance a state.

States as Extension to Language Descriptions

The actual language description and state language description are closely related. Since language instances are part of initial states, the state language description contains all the elements of the language description. The state language description is an extension of the language description. These extensions compass descriptions for all the runtime state information that is necessary to describe the dynamics of a language instance.

Lets take Petri-nets as an example. Fig. 4.3 shows a meta-model for place-transition-nets. Each instance of that meta-model is a Petri-net. Fig. 4.4 shows two popular instances of the Petri-nets language: a semaphore on the left, and the dining philosophers on the right.

You cannot use this Petri-net meta-model to describe the states of running Petri-nets, because you don't have the means to store the actual number of tokens in a place. The Petri-net meta-model is useful to describe the Petri-net language, but not its dynamics which depends on *token allocation*. A suitable description for a Petri-net state language is presented in Fig. 4.5. This meta-model includes an attribute for places, which can be used to store the number of tokens in a place. Example instances of this language are shown in Fig. 4.6.

The small set of example states shows that each Petri-net can have multiple token allocations. The top row of examples shows typical starting allocations for the modelled problems semaphore and dining philosophers. The second row shows example allocations that can be reached from the allocations above, and the last row shows allocations that you don't want, but that

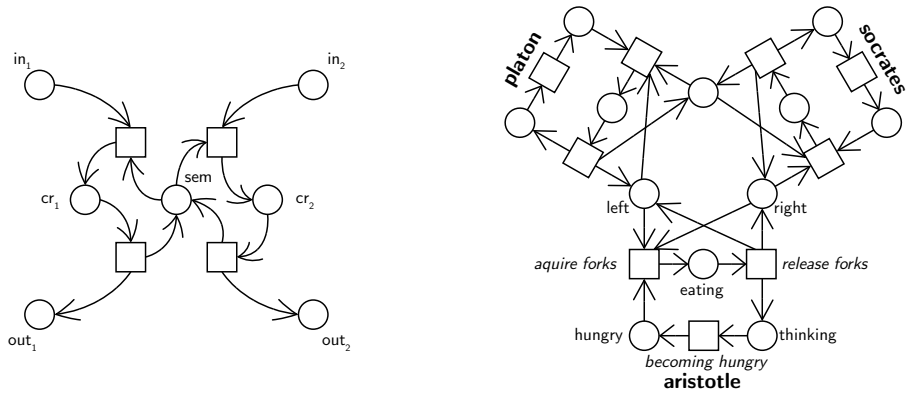


Figure 4.4: Some example Petri-nets: a semaphore and a net modelling the dining philosophers problem.

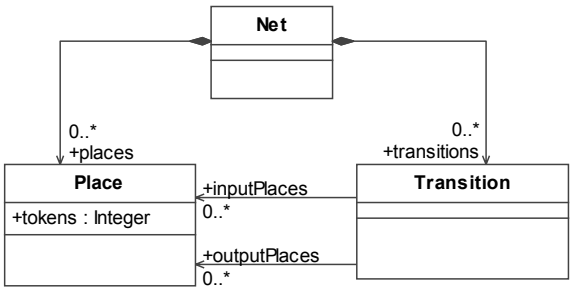


Figure 4.5: A meta-model for Petri-nets including token allocation.

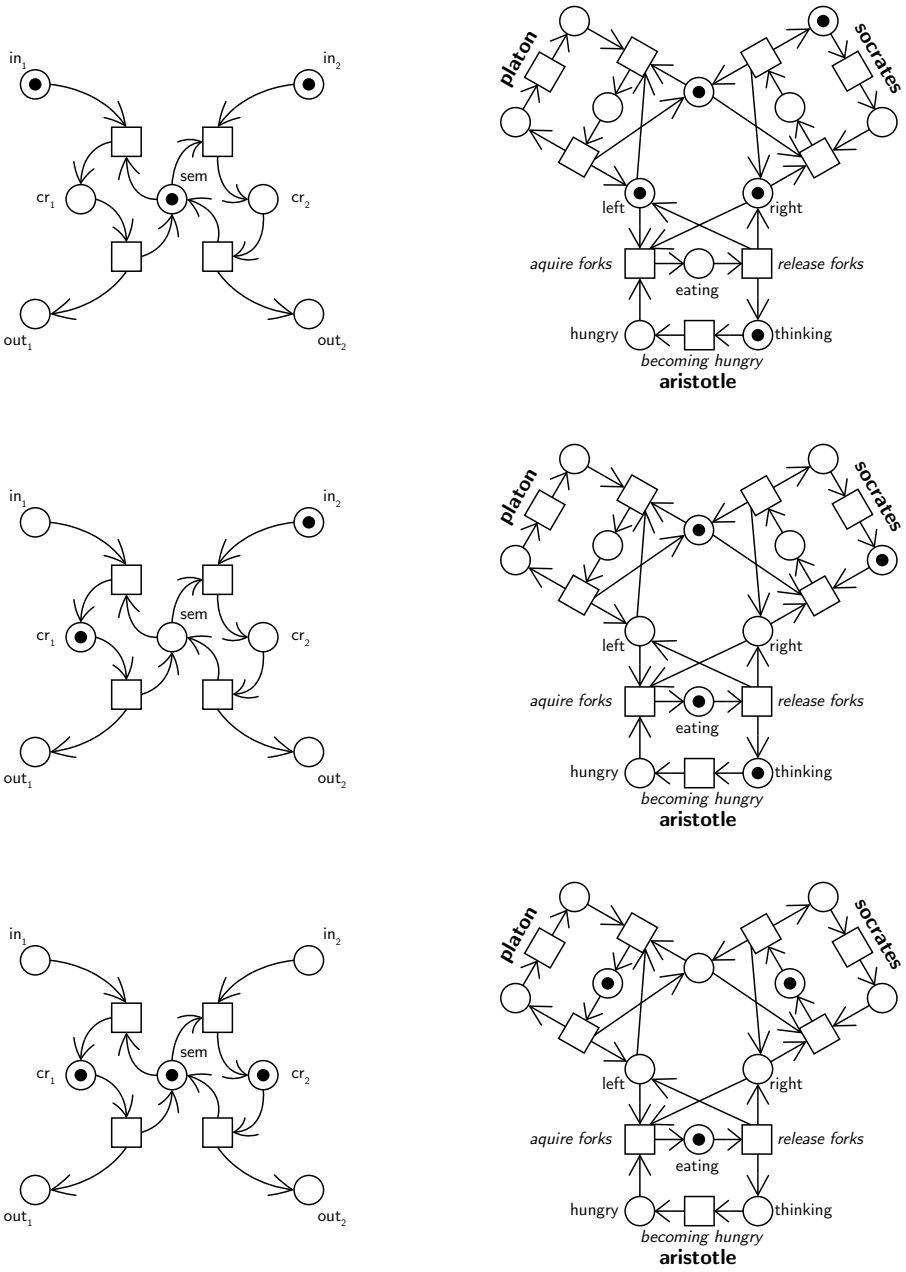


Figure 4.6: Some example Petri-nets with token allocation.

also valid instances of the state language description.

Defining Initial States

A reasonable initial function for Petri-nets would assign all those states to a language instance that are token allocations for this instance. This means that each allocation for a Petri-net is a valid start state for a Petri-net behaviour.

In general, a behaviours initial states must *contain* the language instance that this behaviour is the behaviour of. Otherwise, the behaviour would be independent of the language instance, i.e. it would be the behaviour for other language instances too and therefore hardly reflecting the semantics of this specific language instance.

An initial state can carry more information than only the language instance. This means for the same language instance there can be more than one initial state. This additional information can carry information about the environment that the language instance is executed in, information about user input, things like the initial token allocation in Petri-nets.

In general the language developer has to define an initial function to determine the possible initial states for a language instance. In MOF-based language description such a function could be described with a query formalism like OCL. You can use OCL constraints that verify whether a state language instance is a valid initial instance per se and therefore a valid initial instance for the contained language instance.

4.2.2 Transitions and Constraints on Transitions

Even though you know that states can be described by meta-models, there are still several ways to define possible transitions and system behaviours. I want to briefly introduce three different approaches.

Code-Generation

Based on the language constructs given in a meta-model, a code generator can be used to generate executable code from a model. This code can use a mixture of model and programming language variables/states to define a system state. The executed code describes state changes by changing the model and/or changing variables within the generated program.

Code-generation is often used to realise a language's operational semantics in an ad-hoc manner. Actually, even though it is used to realise operational semantics, the code-generator is rather a description for translational

semantics (from the language to the programming language the code is generated for) than a description for operational semantics.

Action Languages

MOF-like meta-modelling framework define a set of predefined actions. These actions are things like: create an object, set an attribute, create a link, delete an object, etc. These actions describe very small, atomic changes in a model. These actions form a state transition system.

Actions can be scheduled from various forms of action languages. Examples are UML activities or most imperative programming languages. Such languages use expressions over models to create decisions and parameters for the actions. Blocks written in those action languages can be linked to the meta-model on different modularisation levels. One possibility is that meta-model operations are implemented in an action language. In this case one operation implementation can call other operations. The whole semantics descriptions feels and behaves like a normal object-oriented program. The other possibility is to provide a behaviour implementation for classes, where each object (class-instance) runs its behaviour once it got created/instantiated. Another possibility is to provide behaviour implementation for the whole model.

Either way, the action language works as a description language for a constraint function. Given MOF actions as a state transition system, an UML activity for example, would constrain behaviours of this transition system to those that execute actions as described in the activity. The same holds for all other action languages.

This approach uses state transition systems that are independent from the actual meta-model. You can define state transitions systems based on actions over the meta-modelling formalism (transitions), based on this formalism only. To describe an operational semantics with this approach, a language developer provides only a meta-model (states) and a constraint function described in an action language (constraints). All the information about a language instance behaviour is captured within the constraint function description.

Model Transformations

Another approach is model transformation. You can define state transition systems with transformation rules. For each pair of states that are source and target model for a model transformation rule, this rule describes a possible transition between these two states. Instead of single rules, you can also

use sets of transition rules to describe transitions in a similar way. Using a transformation language, one can define more complicated state transitions than with simple actions. Model transformation allow more complex, language-dependent transitions between states.

For rule-based transformations also several ways of scheduling exists. For example, transformations can also be choreographed using action languages, where the execution of a transformation rule becomes an action. Transformation rules can also be grouped with logical operations: one rule can only be applied when another rule could already be applied, or several rules that can be fired are alternatives to each other, where one of the rules is either selected non-deterministically or via rule prioritisation. The different ways of scheduling can be used to describe constraint functions for the transitions systems defined through the used transformation rules.

Using this approach, a language developer describes both, transition system and constraint function, depending on the language, even though a constraint function might not be necessary in all cases. A language developer has to provide meta-model (states), transformation rules for model-to-model transformations within the meta-model (transitions), and a schedule for the transformation rules (constraints).

4.3 Problems

Automatic model execution

Today's techniques for semantics description are only tailored for human readable language descriptions. The involved meta-languages often neglect many critical details like input, output or platform bindings. This is good enough to understand a language on an abstract level, but not enough to execute a language instance without manually adding further details.

This calls for an model-driven architecture approach separating platform-independent and platform-dependent semantics descriptions. The problem is that you want many things at a time:

- You want a short abstract easy to understand description of the language semantics.
- At the same time you want to be concrete and complete enough to execute models.
- And you want it to be very flexible, reuse existing functionality, and run on existing platforms.

- You need to relate basic concepts to a platform and do not describe these concepts explicitly. These concepts are: time, concurrency, interaction with an environment.

One solution could be to integrate multiple formalisms to define parts of the semantics of a language. So you can choose from different abstraction levels for different parts of the semantics descriptions. Different techniques can interact by using the same representation for states. The idea is to use the best of both worlds: abstraction where possible, details were needed.

Formality

Semantics description formalisms are usually tailored for a specific purpose, which is either a mathematical formal description of the language semantics (behaviour representations instead of actual executions or simulations) that allows to proof certain language or language instance properties, or a computer understandable description that allows language instance execution (execution or simulation as behaviour). There are no meta-languages that can be mapped to both: a semantic domain with mathematical behaviour representations and a semantic domain consisting of executions or simulations.

Mathematical formal descriptions are usually written on an abstraction level that is to high. Many technical details are omitted that would be necessary to automatically create a realisation for this semantics. If a mathematical formal description would allow to express all these details, it becomes far more complex, and despite its mathematical nature, it would not allow to proof language properties with reasonable efforts.

Sheer size

Languages can be very big, and semantics descriptions can be even bigger. You need mechanisms to structure semantics, reuse different pieces of semantics descriptions.

Tool integration

Languages do not only consists of semantics. The semantics description and derived tools like simulators or interpreters must co-work with other descriptions and corresponding tools like notations and editors.

4.4 Related Work

The description of operational semantics, especially for programming languages, has been a major issue since the early days of programming languages. The basic structure of an operational semantics description was introduced by Plotkin [85]. Classical mathematical formal description techniques include denotational semantics, proclaimed by Scott and Strachey [113, 109] or axiomatic and algebraic semantics descriptions.

The *Ptolemy* (and *Ptolemy II*) project [61, 122] combines several small modelling languages for the design, modelling, and simulation of real-time, embedded systems. Thereby, Ptolemy targets several semantic domains in a cohesive framework. This allows Ptolemy to model heterogeneous systems that span several domains (e.g. a system that is partially described through discrete events, and partially through continuous time processes) [11]. Although the used languages are not automatically developed, they are described. This work resulted in a description and classification of several semantic domains interesting for system design languages: DE (discrete event), CSP (communicating sequential processes), DDE (distributed discrete event), PN (process networks), and CT (continuous-time).

Nowadays, researchers go beyond formal description techniques and furthermore claim immediate machine-based executeability of operational semantics description. This is archived either through interpretation of semantics descriptions or through generation of language instance interpreters or simulators from semantics descriptions. One especially practical mathematical formalism are Abstract State Machines [41, 9]. These are computationally easy to handle, and several programming languages that realise the ASM formalism have been developed. With these ASM-based languages, operational semantics descriptions written in ASM can be turned into prototypical or reference interpreters or simulators with low manual work necessary. This was, for example, done with the formal semantics specification of SDL [30].

Along the current DSL movement, many frameworks for the DSM of interpreters and simulators have been created. Such frameworks exist for both MOF-like meta-model-based language descriptions and grammar-based language descriptions. Furthermore, these approaches can be categorized into approaches using action languages and approaches using transformations. Action languages are used to describe the behaviour of language instances based on a structural language description, i.e. meta-model or grammar, while transformations are used to define state transitions, where possible states are described with either meta-models or grammars.

Frameworks based on action languages are AMMA [47], Kermeta [115], XMF [12]. In [92] Abstract State Machines (ASM) are integrated into the

DSL framework AMMA to support specification of execution semantics for DSLs, using ASMs as an action language. Muller et al. [68] use a textual action language in combination with OCL for high level semantics descriptions. This action language is executable and provides the foundation for the DSL framework Kermeta. A similar approach is used in XMF Mosaic: it uses an OCL version extended with actions to define language semantics.

Frameworks that use transformations to execute language instances include EProvide [98], GME [21], or Atom3 [22]. Sadilek and Wachsmuth created EProvide [98]. It uses QVT-based model transformations to describe and visualize operational semantics for EMF-based languages. In [50] Karsai et al. use their framework GME in combination with graph transformations to specify model interpreters. De Lara and Vangheluwe use graph grammars and their framework Atom3 [23] to define the operational semantics of a visual modelling language. Engels et al. describe the operational semantics of UML behaviour diagrams in terms of collaboration diagrams which represent graph transformations.

Very similar to the presented approach, language descriptions with *Story Driven Modeling* (SDM) and *Story Diagrams* [32] use MOF-like meta-modelling and UML activities to describe semantics. SDM does not use predefined actions, but allows to describe runtime state changes via graph rewriting. The *Fujaba* framework allows to edit meta-models and story diagrams and to generate Java code that realises language instance execution.

During the work on this thesis, a series of operational semantics description frameworks based on MOF-like meta-modelling have been created. Fig. 4.1 lists these frameworks and some of their features for comparison. I only compare frameworks that are based on Plotkin's scheme and use a more or less imperative behaviour description language. Other frameworks, e.g. based on graph transformations, are too different to be compared based on shared framework features. In this table, I cover the meta-modelling language used to describe system states, the different languages that can be used to describe behaviour, whether syntax and runtime constructs can be explicitly separated, whether the supported behaviour description language is graphical, and whether it is possible to debug the semantics description.

The first row shows *MOF Action Semantics* (MAS) [105], the framework built for this thesis and presented later in this section. M3Actions [112] and eProvide [98, 100] have been created after MAS, based on ideas covered in this thesis. The meta-modelling framework MOFLON [6] combined with *Story Driven Modeling* (SDM) [32, 4] allows to choreograph model transformations with activities to pass through system states; kermeta [115, 68] is a meta-programming language that can be used to implement meta-model operations similar to Java. With MAS, I wanted to create a framework

	meta-modelling	languages	syntax-runtime sep.	graphical	generative/interpretative	debug
MAS	CMOF	Activities+OCL, Java	yes	yes	int.	no
M3Actions	emf	Activities+OCL	yes	yes	int.	yes
eProvide	emf	Java, QVT, ASMs, Prolog	no	-	int.	yes
MOFLON	CMOF	Activites+graph trans.	no	yes	gen.	no
kermeta	km3	kermeta	no	no	int.	no

Table 4.1: MOF Action Semantics (MAS) in comparison with other operational semantics description frameworks based on framework features.

that uses existing and widespread (graphical) modelling languages, opposite to graph transformation in SDM, and a completely new language in kermeta. MAS uses a clear separation between modelling syntax and runtime constructs (semantics), which distinct it from the other approaches (except M3Actions). On the downside, MAS lacks in performance compared to its competitors, due to additional overhead in calculating changes to redefined and sub-setting properties (possible in CMOF meta-models, see 2.5) and its interpretative nature.

4.5 An Action Language for MOF

This section presents a method to describe the operational semantics of languages based on a MOF-like meta-model. I extend MOF's meta-meta-model with additional constructs that allow you to describe language instances and runtime states within one description. In such a description you can relate runtime constructs with language constructs. A description for language constructs and runtime constructs is basically a meta-model and each model constitutes a system state. Each meta-model instance without instantiations of runtime constructs is a possible initial state. Furthermore, I define a state transition system based on atomic modifications that can be applied to a

model in MOF. Each possible model manipulation (creating a model element, deleting a model element, changing a property value) defines a transition between two models (states). Of course, this simple state transition system creates an unreasonable general behaviour. I present several meta-languages that can be used to describe constrain functions to finally describe the wanted operational semantics.

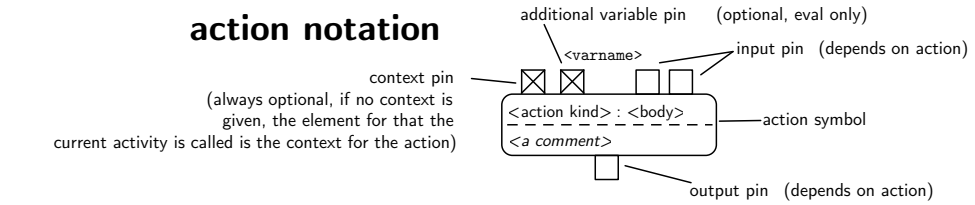
While you use MOF to describe language and runtime states, you can use several meta-languages to describe the specific behaviour. This can be multi-purpose programming language like Java, or languages on a higher-abstraction level, such as ASMs. In this thesis, I want to use a combination of UML activities and OCL. This combination promises a series of advantages: Activities and OCL are well known languages that many potential language developer are familiar with; I use a visual language which is easily comprehensible for human beings and might also be understandable for non engineers like the usual domain expert, and Activities and OCL operate on a high-level of abstraction.

Nevertheless, the choice of language for describing the language dynamics is always a trade-off between high abstraction and expressiveness (Activities/OCL) and flexibility (multi-purpose programming language). Therefore, I design the semantics description meta-languages in a way that allows you to combine both choices in a single description method.

Section 4.5.1 explains the basic concepts of my method and shows an example language description, which describes the operational semantics of Petri-nets. In section 4.5.2 I show that more complex languages need to distinguish between language constructs (syntax, they define the models that the user can write) and runtime constructs (that describe additional information that is necessary when a model is executed). Section 4.5.2 discusses reusable designs for operational semantics and presents a pattern for instantiation as an example.

4.5.1 Basic Concepts

As you learned, operational semantics describes transitions between models (states). Such transitions can be realised by changing a model. To describe and execute operational semantics defined with such transitions, you need: (1) changeable models, (2) possible transitions, and (3) a language to constrain what transition is to be executed under what conditions and in what order. You have all these things: (1) MOF models can be maintained and manipulated within a model repository; (2) MOF defines primitive actions that can be applied on models; (3) you can use all sorts of imperative languages to describe the execution of action after action.



actions (as symbol with possible parameter)	**description**	**semantics** (most actions are already defined in the MOF standard, we refer to the according MOF operations defined in CMOFReflection)
<varname> eval: <OCL expression>	Evaluates the given OCL expression	Based on OCL. The context determines the value of *self*. Additional variables can be used in the expression. The value that the expression represent is provided through the output pin.
call: <operation>	Calls another operation.	Based on MOF. The context determines the object on that the operation will be called. Input pins determine the used arguments. Operation return arguments are provided via output pins. Input/output pins (grey) depend on the operation. `Object::invoke(op: Operation, args: Argument[0..*]): Object[0..*]`

create: <class>	Creates a new instance.	Based on MOF. The given class has to be a metaClass of the used meta model. The result element is provided via the output pin.
		`Factory::create(metaClass: Class): Element`
		If this action is used on a class that is a *runtime representation of* another class, a link is created between the context and the created instance according to the semantics of *runtime representation of* (see section 4).
		`Factory::createLink(association: Association, firstElement: Object, secondElement: Object): Link`

set: <property>
set: <qualified property>
	Sets the value of a property with multiplicity 1 or 0..1. For properties with qualifiers additional input pins are required	Based on MOF. The context determines the element that will be changed. The input pin determines the used value. If more than one input is used, the first input pins determine qualifiers.
		`Element::set(property: Property, value: Object)`
		`Element::set(property: Property, qualifiers: Object[0..*], value: Object)`
		`// an extension to MOF`

remove: <property>
add: <property>
	Adds/removes an value to/from a property with multiplicity bigger than one.	Based on MOF. The context determines the element that will be changed. The input pin determines the used value. The values of the property in the given elements are retrieved with:
		`Element::get(property: Property): Object`
		The result of this operation is assumed to be a collection. The value is added to or removed from this collection.
		`ReflectiveCollection::add(value: Object):boolean`
		`ReflectiveCollection::remove(value: Object):boolean`

Figure 4.7: A list of actions that can be used to define operational semantics.

As said, an operational semantics for a language L is a tuple $S = (L, \Gamma, \rightarrow, c, \mathit{initial})$. A MOF meta-model describes a set of models. Each model can be seen as a system state: I use a meta-model to describe Γ. Because the state describing meta-model contains descriptions of language constructs and runtime constructs, only those meta-model instances that only contain language construct instances form the set of language instances L. The state transitions are defined by the predefined primitive MOF actions for models. The *initial* function assigns each element in L the same element in Γ, which is trivial because $L \subseteq \Gamma$. I use a language based on UML activities to describe the constrain function c. When an activity is executed, it executes actions, which leads to a sequence of states. Therefore a running activity selects a specific state sequence from the general behaviour. If activities can also execute other activities, sequentially, as well as, concurrently, you can describe a constrain function that selects the specific behaviour that you want to describe.

Models Models, as instances of meta-models, are normally not supposed to change. An UML model, for instance, does not change once it is written. But because models constantly change during editing, MOF-like frameworks already support model changes. You can dynamically create new elements or update attributes of existing elements to change a model.

Atomic actions Fig. 4.7 defines a fixed set of atomic actions. This figure also includes a concrete syntax for these actions as used in the UML activity-based language. The semantics of some of these actions lead to a manipulation of the model, i.e. a state change. Other actions call activities declared as operations, or evaluate OCL expressions over the model. These actions do not change the model and do not account to the state transition system, but they are necessary to create property values, as input for conditional branches, or to call other activities.

UML activities The semantics for UML activities (as I use them) is founded on Petri-nets as described in [114]. Activities are connected to the meta-model via operations. The behaviour of each operation in a meta-model can be implemented with an activity. When a operation is called, the corresponding activity is interpreted. Meta-models are object-oriented models, and calling a operation means that it is called on an instance of the corresponding class. This also means that activities are always interpreted in the context of an object. This context can be addressed with the value *self*.

A meta-model can contain several operations implemented by several activities. This allows to modularise the semantics descriptions. Operations can have parameters, and calling an operation requires corresponding arguments, which can be used in the activity. An advantage of this operation-based distribution of semantics description is that a model can be executed like a normal object-oriented program by calling an operation and interpreting the corresponding activity. Because meta-models are object-oriented, designing a language and its operational semantics becomes a object-oriented design task, and general object-oriented design knowledge and techniques (e.g. design patterns) can be applied.

One operation has to serve as a dedicated *main* operation. There is usually a model element, known as the outermost composite, which contains all other elements. It is reasonable practice to define the *main* operation in the class that describes this element.

An Example Language – Petri-Nets

In this section, I demonstrate my meta-modelling method and create a language model for Petri-nets. This model consists of descriptions for the language constructs (syntax) and an operational semantics for Petri-nets. Fig. 4.8 shows the language description as meta-model and an example Petri-net. I choose Petri-nets as an example language, because they have a very small set of language constructs and simple but clear semantics.

The language description specifies that a Petri-net consists of places and transitions. Places can be related to transitions, and each transition has an arbitrary number of input and output places. A place can contain any number of tokens. The figure also shows an example Petri-net, an instance of the given meta-model. This Petri-net diagram of the famous dining philosophers uses the typical Petri-net notation: places are drawn as circles, transitions as boxes. Incoming arcs show the input places of a transition, and outgoing arcs show their output places. Dots inside places show the number of tokens in a place. All text in this Petri-net diagram is commentary, and no text fields are defined in the meta-model. However, I will use these names in further explanations. Please note that I mixed Petri-net structure (places and transitions) with Petri-net configurations (tokens). I will address this issue in section 4.5.2.

The semantics of Petri-nets is simple. Transitions are the only active elements in a net. They change the number of tokens in places, which are the only dynamic elements in a net. A Petri-net transition changes the number of tokens in its input and output places when it is fired. But a transition may only be fired when it is enabled, and it is enabled when all its input

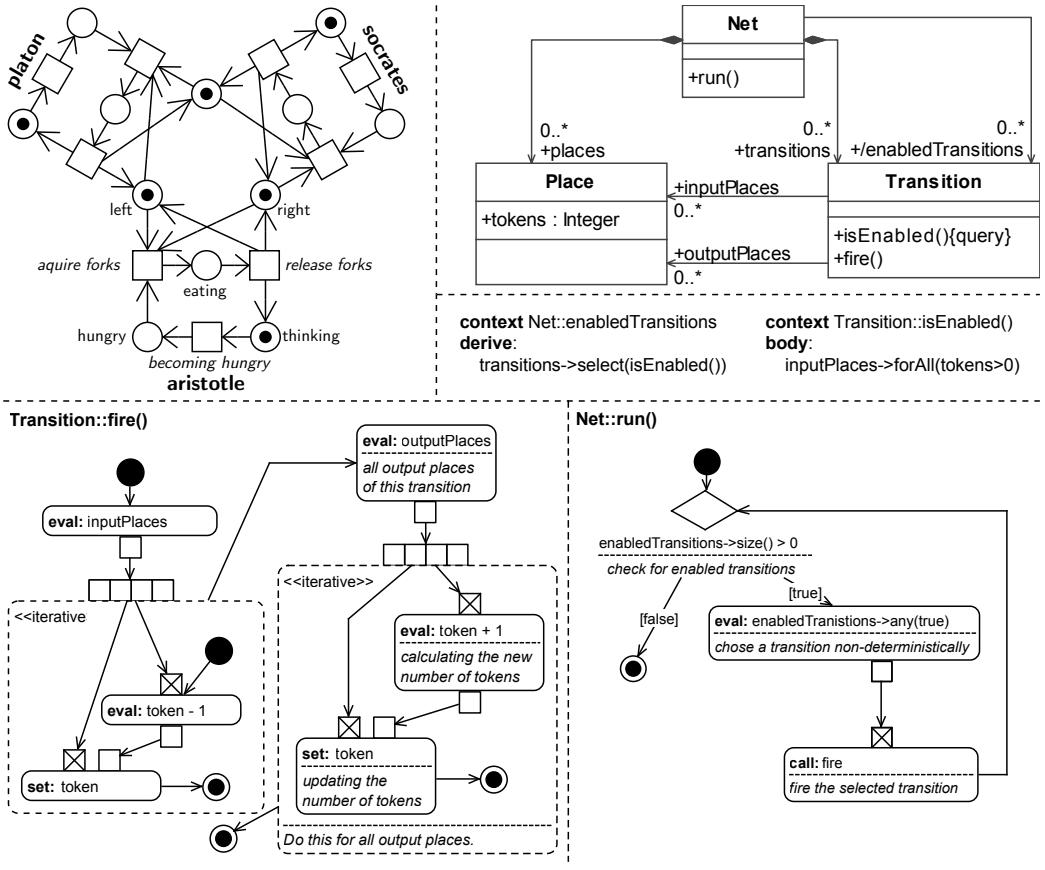


Figure 4.8: Petri-nets as an example: an example net and a language model for Petri-nets containing an abstract syntax model, OCL expressions, and activities.

places contain at least one token. Given these definitions, a Petri-net has the following semantics: a transition is chosen from all the enabled transitions non-deterministically. The chosen transition is fired. This means that the number of tokens in all input places is reduced by one, and the number of tokens in all its output places is increased by one. Transitions are chosen and fired until the net contains no more enabled transitions.

You describe the semantics with operations and derived properties. The Petri-net meta-model contains two operations, one query operation, and the derived association end `enabledTransitions`. These elements have to explain the informally given semantics in a formal and executable way. The queries and derived property can be determined by OCL expressions. These elements need no further refinement or implementation; the OCL expressions can be evaluated by the computer right away. `Transition::isEnabled()` returns *true* when the transition contains tokens in all input places. The derived association end `enabledTransitions` selects the collection of all enabled transitions in a net. The OCL expressions are given in fig. 4.8.

The behaviour of the other two operations can be specified using the activity language (see fig. 4.8). Imagine that the operation `Transition::fire()` is called for the transition *becoming hungry* during the execution of the example net. The first action is to evaluate the expression `inputPlaces` in the current context *becoming hungry*. This transition has only one input place: *thinking*; the result is a collection containing *thinking* only. After that, the collection is iterated. The activity in the iterative expansion region is executed for each element; in this case this is only *thinking*. This sub-activity evaluates `token`-1. This time *thinking* and not *becoming hungry* is used as context. The value $(1 - 1 = 0)$ is the result and is set to the property `token` in the context of *thinking*. After that is done, the number of tokens in each output place is increased in a similar fashion.

The operation `Net::run()` acts as *main* operation; it executes the net. This means it fires enabled transitions as long as there is at least one enabled transition left. A decision is used to continue or stop based on whether the set of `enabledTranitions` is empty or not. When it is not empty, one transition is selected non-deterministically, using OCL's `any`. After that, `Transition::fire()` is called on the selected transition.

4.5.2 Distinguishing Between Syntax and Runtime Elements – Instantiation

In the last section, I defined operational semantics by changing a language instance. All runtime information needed to execute a language instance

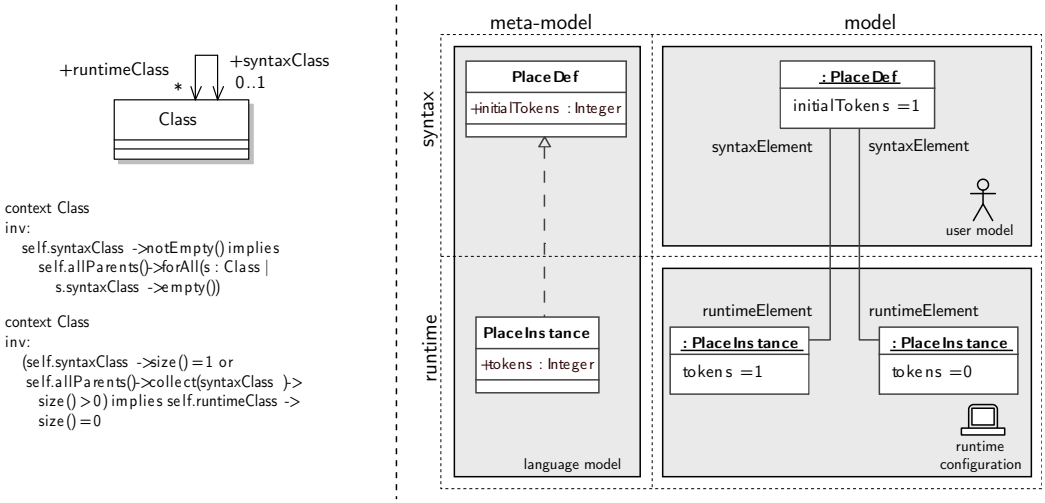


Figure 4.9: A new meta-model concept to relate syntax and runtime elements with each other.

could be stored within the instance. This approach has two flaws. One, in general you need additional data structures to describe runtime states. A program, for example, is only one part of a configuration during a program run. Other parts are slots for variable values, heap memory, and call stack frames. The second problem is that when you change the language instance it will be lost for future execution. In the moment you destroy a token in one Petri-net place and create it in another, you destroy the original marking. When you say the initial marking is part of the Petri-net, you would destroy the net by executing it. You should have stored the actual number of tokens independent from the initial number of tokens. To describe complete states, a meta-model has to define both: the constructs that built language instances and additional constructs needed for runtime information. I distinguish between syntax classes and runtime classes. The set of all syntax classes describes what users of the language can write in their language instances. The set of all runtime classes describes data that can be created and used during the execution of a language instance. Syntax and runtime elements can be related to each other.

Fig. 4.9 shows (on the left side) an extension of the MOF meta-meta-model as a meta-model for a new relationship between classes. I call this relation *runtime representation of*. This directed relationship indicates that one class denotes a runtime representation of a syntax class. I use the UML realisation arrow (which has no predefined meaning in MOF) to notate this relationship. Fig. 4.9 also shows two corresponding OCL constraints that limit the use of this relationship: there are no circles allowed and a class

cannot be runtime representation for itself.

The right side of Fig. 4.9 shows an example of a runtime representation: a `RuntimePlace` is the runtime representation of a `PlaceDef`. The language user, who creates the Petri-net, determines the initial number of tokens using the corresponding slot in `PlaceDef` instances. At runtime, the numbers of tokens are stored separately in `RuntimePlace` instances. That allows you to run the same net in two runtime representations at the same time. The *runtime representation of* relationship will allow to navigate between instances of runtime and corresponding syntax classes. The `create` action described in the previous section, will automatically link a newly created instance of a runtime class with the corresponding syntax class instance.

An Advanced Example Language – Hierarchical Petri-Nets

In this section, I use a more sophisticated Petri-net variant to demonstrate that most semantics descriptions require to differentiate between syntax and runtime elements. In the previous section, I modelled a dinner table with three philosophers. This model already contained the same philosopher pattern three times. I now describe *Hierarchical Petri-nets* (also known as *modular Petri-nets*, not to be confused with Petri-nets that use sub-nets as tokens), which allow to build an abstraction for this pattern. You can model the common philosopher behaviour once, and use it for multiple philosophers. Fig. 4.10 shows such a hierarchical Petri-net for the dining philosophers.

Hierarchical Petri-nets contain additional constructs and notations. You can define sub-nets, notated as a smaller net inside a box. These sub-nets have dedicated interface places. In the example the behaviour of a philosopher is modelled as a sub-net. The places for his left and right fork are interface places, because each philosopher has to share this place with his right and left neighbour. In hierarchical Petri-nets each net can contain sub-net uses which are notated as a black box. Petri-net uses are related to regular places to connect interface places with real places. These connections are drawn with lines that have the respective interface place name written on them.

Since hierarchical Petri-nets contain additional constructs, you also need a different language meta-model (fig. 4.11) with additional classes and different descriptions of operational semantics. You have to distinguish between the definition of a sub-net and the use of a sub-net. `NetDef` represents Petri-net models. `NetDef` instances can contain transitions, sub-net definitions, sub-net uses, and places. Net uses are realised in the class `NetUse`. Instances of `NetUse` reference a `NetDef` to characterise the used sub-net. The former class `Place` is now called `PlaceDef`. Instances of this class are used to model

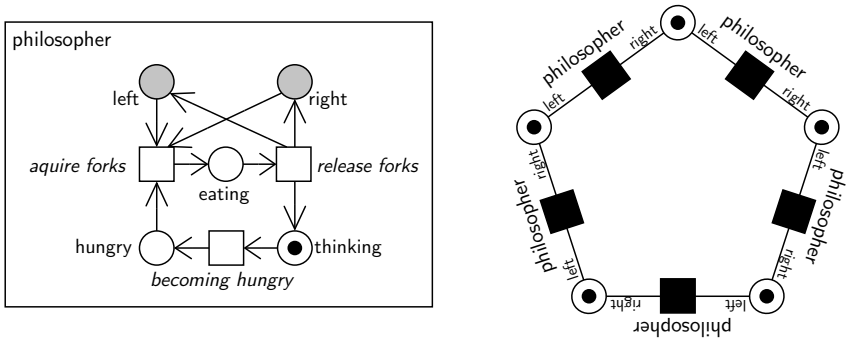


Figure 4.10: A hierarchical Petri-net for the dining philosophers.

places; you will need another place class to represent places at runtime. The connection of interface places is modelled as a qualified property of `NetUse`. A qualified property works like a map. In this case, it associates a `NetUse` with a `PlaceDef` based on another `PlaceDef` as key: a use is connected to places, and each of those connections is qualified by an interface place.

Hierarchical Petri-nets use one sub-net several times. You have to use several instances of the same net definition to to store the number of tokens in each sub-net instance separately. You cannot use the semantics description from the place/transition Petri-net example, because the places in one sub-net are now used several times in multiple uses of the same sub-net. When the number of tokens in a place of one instance changes, it would also change in the same place of all the other instances.

The definition classes `NetDef` and `PlaceDef` are syntax classes. They are used within the Petri-net model; they are classes for things the user draws in a Petri-net diagram. The dining philosopher model is a `NetDef` instance, the philosopher sub-net is a `NetDef` instance; all places in the model are `PlaceDef` instances. The other two classes `RuntimeNet` and `RuntimePlace` are runtime classes. A `RuntimeNet` can contain instances of sub-nets (other `RuntimeNet` instances) and contains `RuntimePlaces` using a qualified property with the corresponding `PlaceDefs` as keys.

When a user provides a hierarchical Petri-net it will only contain instances of the syntax classes. Creating runtime class instances is part of the semantics. It is part of the semantics to initially instantiate the dining philosophers Petri-net, create sub-net instances for all the uses of philosopher. This instantiation task is modelled in the operation `NetDef::instantiate`. This operation will create a runtime representation of itself and all its contained places; it will furthermore create runtime representations of all used sub-nets recursively and connect its interface places to real places. Fig. 4.11 shows the activity diagram for this operation.

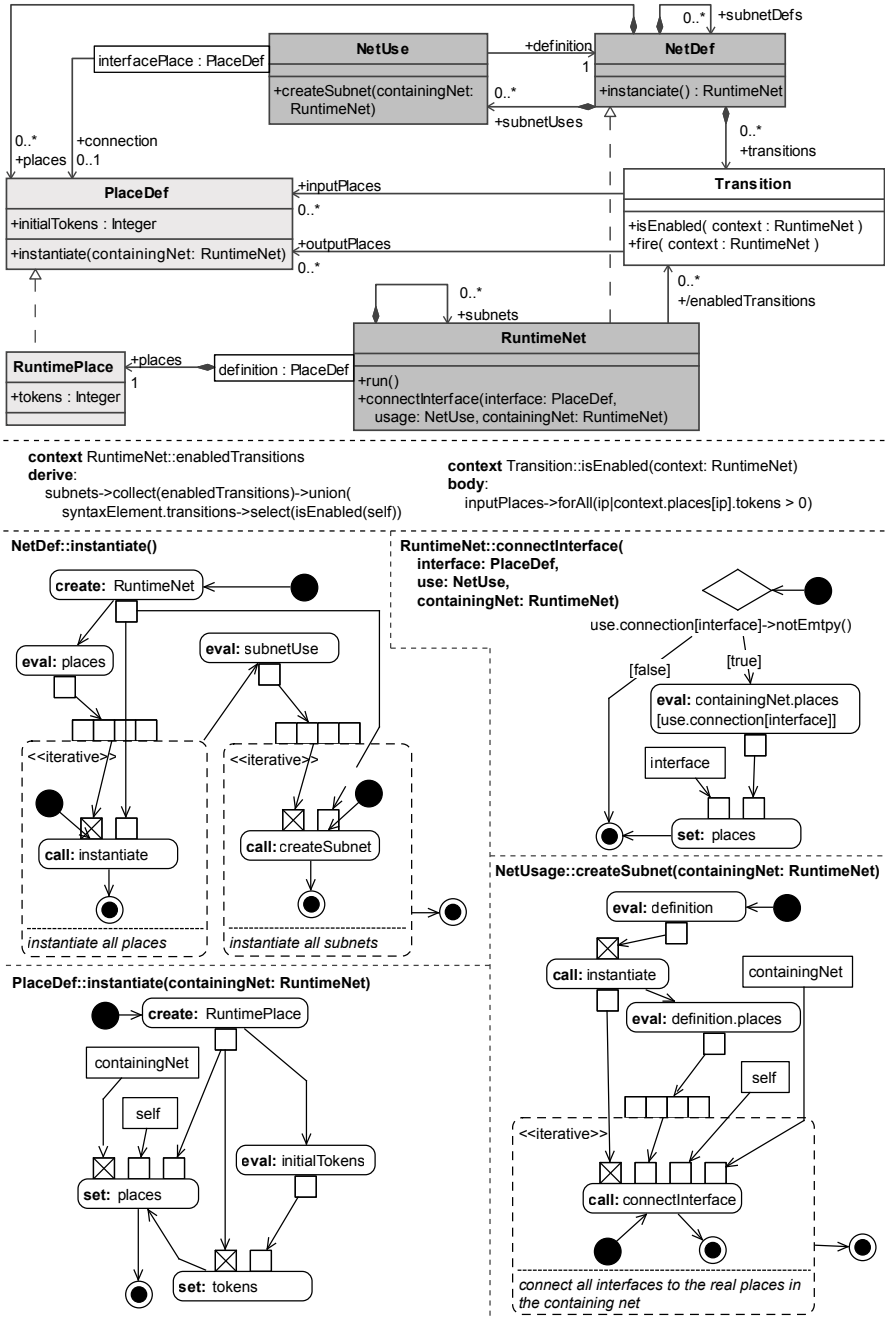


Figure 4.11: A language model for hierarchical Petri-nets.

After `NetDef::instantiate` was called for the top-level Petri-net, you can use the created `RuntimeNet` by calling its `run` operation. Even though `run`'s signature hasn't changed from the previous section, it works a little different due to the changes in the meta-model. Transitions can only be fired in the context of a `RuntimeNet`. Since transition is only a syntax class with no runtime counterpart, it is also only related to `PlaceDef` (the syntax class for places). The input and output places of a transition are instances of `PlaceDef` and the number of tokens cannot be accessed or changed directly on them. The operations of transition have to access the corresponding `RuntimePlace` using a `RuntimeNet` as context. The run operation itself (not shown) also works different: it still chooses one transition from all enabled transitions. But because one transition can be enabled in different sub-nets (in the starting configuration, *becoming hungry* is enabled in all five philosophers) `run` must also chose one of these sub-nets that the chosen transition is enabled in. After transition and `RuntimeNet` are chosen, run fires the transition using the chosen `RuntimeNet` as context argument.

A General Instantiation Pattern

Fig. 4.12 shows an abstract pattern for instantiation (white classes) and how it is used by specialisation in two examples (grey and dark grey). This pattern describes the abstract concept instantiation. It defines `Classifiers`, which define sets of instances with common attributes defined as `Features`. These features can have a type. A `Classifer's Instance` provides a `Slot` for each `Feature`. Each `Slot` can hold `Values` of the corresponding `Type`. Implementations of operations `Classifier::instantiate`, `StructuralFeature::instantiate`, and `Create::create` are written once and then reused. The implementations instantiate the corresponding runtime constructs dynamically. `Classifier:: instantiate` creates an instance of the corresponding specialisation `Instance`; it used `StructuralFeature::instantiate` to create a slot for each classifier feature and puts these `Slot` instances into the property `slot` qualified by the corresponding structural feature.

This pattern is common to many languages, including MOF and UML. Without knowing it, I already used this pattern in the previous section, where I had `NetDef` (classifiers of sub-nets that can be instantiated at different places) and its runtime counterpart `RuntimeNet` (runtime representations of sub-nets). `NefDefs` have `PlaceDefs` as features and `RuntimeNets` have `RuntimePlaces` as corresponding slots. You don't need the type/value part of this pattern, because the number of tokens is always stored as an integer. But here you could extend the language if you wanted to introduce objects for tokens as in *Object Petri-nets*.

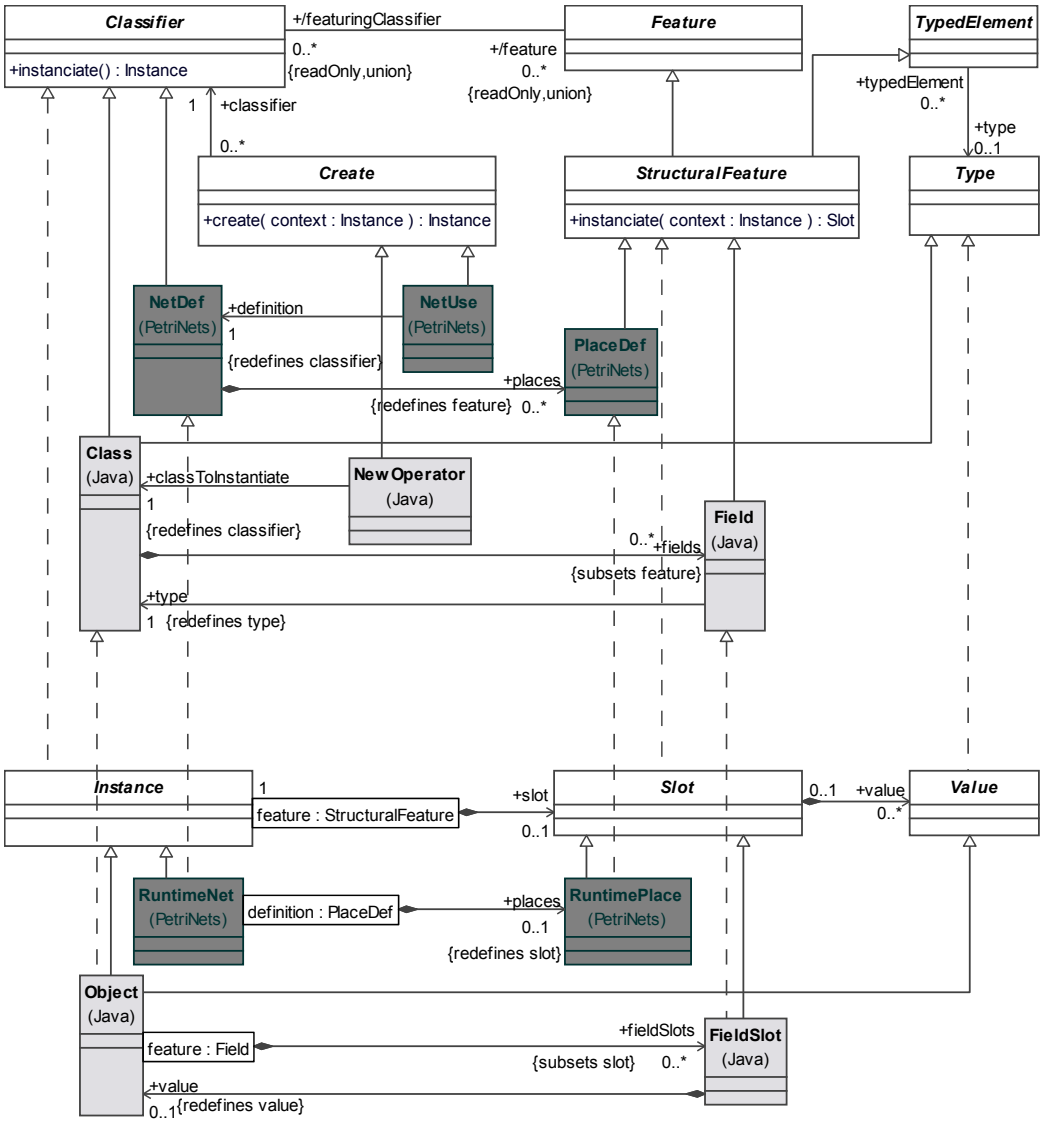


Figure 4.12: A general pattern for classifier and instances.

The other example application for this pattern is `Classes` and `Fields` (also known as member variables) in object-oriented languages such as Java. This application of the pattern also uses `Types` and `Values`, even though I simplified the problem in this example: classes are the only types and conclusively objects the only values. Another application for this pattern are procedure-like concepts. Procedures are classifiers, parameters and local variables are features, call frames are procedure instances with proper slots for local variable or parameter values.

4.6 Conclusions

After introducing a general framework for describing operational semantics based on state transition systems, I presented a meta-language to describe such semantics. For this meta-language, I built a generic interpreter that takes a language description, including the semantics description, and an initial configuration (i.e. language instance) as input. It executes the operational semantics description on that language instance. During this execution the initial configuration is changed (transitioning based on simple atomic actions) according to the given semantics description. This allows you to execute language instances solely based on a language description.

The presented meta-languages and meta-tool provide an efficient DSM approach to building interpreters for language instances. Language developers can model the described language semantics on a high abstraction level with a visual language and execute language instances right away. However, this approach has a serious of weaknesses and unsolved issues.

Firstly, you only describe the semantics by means of changing a runtime configuration, but many semantics need to interact with an underlying platform, which is not directly accessible from your semantics description. Any platform objects or functionality that the semantics uses, needs to be integrated into the runtime configuration meta-model to be used. The current state of the art would be to create some configuration classes with operations implemented in a multi-purpose programming language (in the case of my meta-tools that would be Java), and use the implementations of these operations to access the underlying platform indirectly. This can be facilitated with meta-model libraries for specific platforms, in the same way that platforms are usually accessible through normal programming libraries.

Secondly, the level of abstraction of the presented meta-language is higher than those of multi-purpose programming languages, but still lower than those of many formal operational semantics formalisms. This bears advantages and disadvantages. On one side, my approach is more flexible than the

many formal approaches, which are often limited to a specific kind of language. On the other hand, my meta-language uses very small actions, which render more complicated transitions between configuration very verbose and cumbersome to write.

Thirdly, I built a serious of tools to create semantics descriptions and to execute them on language instances. While these tools are convenient (and mandatory) to create the descriptions, they lack in the ability to analyse semantics descriptions. There are a lot of warning and error conditions thinkable that an corresponding meta-tool could find. But finally, this is just a matter of tooling maturity. Another tooling aspect are debuggers. When you want to interpret a language instance, you usually also have a use-case for debugging language instances. Furthermore, you need to debug the semantics description as well.

Projects of different scientists that carry on the research presented here, address these issues. There are three projects that continued the research presented in this chapter: ESemantics [96], EProvide [98], and M3Actions [112].

ESemantics is based on Eclipse EMF and scheme. It uses EMF to describe language and runtime states and the programming language scheme to describe transitions and constraints on behaviours. While it basically works like my approach, it can be used for both simulation and interpretation. You can use two slightly different semantics descriptions, one bound to the targeted platform (interpretation) and one bound to a simulation environment. The benefit of ESemantics is that the most of the semantics description can be reused among interpretation and simulation.

EProvide is also based on Eclipse EMF and provides a general meta-tool architecture for semantics description languages. It again uses EMF to describe language and runtime states, but the language used to define transitions (and, or constraints) is not fixed. Different meta-language adapters exist for EProvide. Among them are Java, QVT, ASMs, Prolog, and Scheme. EProvide allows to select a semantics description language based on its specific abstractions. In other words, you can select the semantics description language that is most suitable for the described language. EProvide also provides a debugger that allows to run language instances. The debugger thereby also allows to reverse transitions and run the language instance backwards in time. This allows to observe an execution, find a mistake in the language instance of semantics description, reverse the execution up to a configuration without error, fix the problem in the instance or semantics description, and finally continue execution.

M3Action is another EMF-based operational semantics description framework. M3Action also uses a meta-language very similar to the present UML activity-based language. But, M3Actions provides advanced tooling, includ-

ing a debugger and checkers for semantics descriptions.

Relating to the problems stated at the beginning of this chapter, I achieve language instance execution from semantics description on a high-abstraction level, without further manual implementation. The meta-language is reasonable abstract yet flexible. It allows to modularise and reuse descriptions along the packaging and object-oriented concepts of MOF-like meta-languages. The meta-language is formal enough to allow execution, but its semantics is not rooted in a mathematical formalism, hence does not allow to formally verify or analyse a semantics description. Modularisation and object-orientation allow to deal with semantics descriptions of arbitrary sizes. The corresponding meta-tools are integrated into Eclipse. Based on this platform, interpretation and simulation of language instances can go hand in hand with other aspects like editing language instances.

Future research should address a series of open issues. These are better debuggers, better analysis of semantics description, better mathematical rooted semantics, closer more transparent integration of platforms, integrating things like real time, concurrency, non-determinism and fairness into semantics descriptions.

Chapter 5

SDL Case-Study

In this chapter, I finally apply the presented methods, meta-languages, and meta-tools. The chosen computer language for this endeavour is the *Specification and Description Language* (SDL). I built a meta-model for SDL's constructs, I describe a textual notation of SDL and create an editor from it, and I describe SDL's runtime constructs and develop an operational semantics description that I used to create an SDL interpreter automatically. All in one, I use DSM to create a whole tool-chain for SDL. Many of the details discussed in this section can only be understood, when you are familiar with SDL, its concepts, and preferably its standard. The software projects [105, 104] contain the SDL descriptions presented here as example descriptions.

This chapter is structured as follows. In section 5.1, I discuss why I took SDL, introduce the used dialect of SDL, and describe SDL as archetype for a specific class of computer languages. In the following three sections 5.2, 5.3, 5.4, I go through the three main language aspects and discuss the description of the SDL language, its notation, and semantics. Finally, I draw conclusions in section 5.5.

5.1 About the Used Dialect of SDL

Since several decades, SDL plays a dominant role as formal specification language for functional aspects of distributed systems, especially telecommunication systems. SDL is maintained by the *International Telecommunication Union* (ITU), which led the evolution of SDL from a simple sketch notation to a complex executable language with a rigid formal operational semantics. SDL offers object-oriented constructs to describe system structure by means of hierarchically structured communicating agents. Behaviour

can be described by composite state machines. SDL includes a complex data model and programming language like computation concepts to allow data exchange between agents, and state machines with more expressive transition triggers, guards, decisions, and actions to perform.

Over the decades, SDL accumulated more and more language constructs, which renderers SDL a very complex, construct rich language. As such, the SDL language standard, a basically English text language description, was structured into two different languages. There is an *abstract* version of SDL, which covers the basic constructs of SDL, and a *concrete* version of SDL, which introduces additional constructs. All constructs of the concrete version can be built from abstract constructs. In other words the concrete SDL provides *syntactic sugar* for the abstract SDL. Because both language versions are formally defined by means of context free grammars, they are called *abstract* and *concrete* in the sense of abstract and concrete syntax. In out sense of the notions language, syntax, and notation, the concrete SDL provides rather a notation for the abstract SDL language.

For this case-study, I concentrate on the abstract SDL. I call the language dialect that I want to describe *SDL Reduced* (SDL-R). SDL-R basically includes the constructs of the abstract SDL language. I neglect all additional concrete SDL constructs, and only use this SDL version to supply you with notation for SDL-R's constructs.

SDL is traditionally notated graphically. This notation is a mixture of diagrams with structured, programming language like, text. However, SDL also has a textual notation that can fully replace the graphical one. Furthermore, this notation is defined formally by means of a context free grammar for the concrete version of SDL. In this case-study, I concentrate on SDL's textual notation, even though there is work to combine DSM of SDL's graphical and textual notations outside the scope of this thesis [87].

This case-study does not only provide a proof of concept solely based on the SDL-R language, but also for all languages that use similar concepts. This includes all languages that use similar concepts or concepts that are a sub-set of the SDL concepts. Such languages are: UML and all languages that are UML profiles, languages based on some sort of state machine, languages like Petri-Nets or languages based on Petri-Nets like most process or workflow description languages, and imperative programming languages like C or Java.

5.2 A Meta-Model for SDL

In this section, I describe a meta-model for SDL-R. Thereby, I rather explain the process of creating the meta-model than explaining the meta-model itself.

I discuss a few examples, but the major focus is laid on the used techniques, how they are executed, and what the benefits are.

5.2.1 A Generated First Meta-Model Version

The starting point for the SDL meta-model is the context-free grammar in the SDL standard that describes the abstract version of SDL. Since this grammar contains all the constructs of the targeted SDL-R language, I can simply derive a first meta-model version with the meta-model generation techniques described in section 3.6.

This automated step gives me a very simple meta-model that basically reflects the grammar. This meta-model has no references (non-composite associations) in it, but uses different forms of identifiers to emulate references. This meta-model contains no inheritance between language constructs, and conclusively does not reuse abstract language constructs. This meta-model furthermore is not structured into packages or diagrams, because the grammar is not structured either. Based on best object-oriented modelling practices, this first meta-model version is inadequate.

5.2.2 General Abstract Language Constructs

Before I enhance the generated SDL-R meta-model, I want to look at some general language constructs that are reused throughout the final meta-model. I partially take these abstract language constructs from the UML infrastructure [73], which provides a library of so called *abstractions*. This library is a set of small packages, each containing a single, small and cohesive language concept. Examples are: ownership, namespaces, types and typed elements, classifiers and features, etc. Most of these *abstractions* barely consist of two classes, i.e. `Namespace` and `NamedElement`, `Type` and `TypedElement`, etc. Fig. 5.1 shows an excerpt of the abstraction library.

Besides these rather small and very abstract constructs, I also want a library of more comprehensive concepts. Fig. 5.2, Fig. 5.3, and Fig. 5.4 show abstract construct definitions for the common language concepts *instantiation*, *concurrency and communication*, and *expressions*. Each of these concepts is given in corresponding packages. Each of these packages contains all the classes necessary to define the constructs for the corresponding concept. The packages do not only comprise syntactical classes but also the necessary runtime classes. I also already explained instantiation in more detail in section 4.5.2.

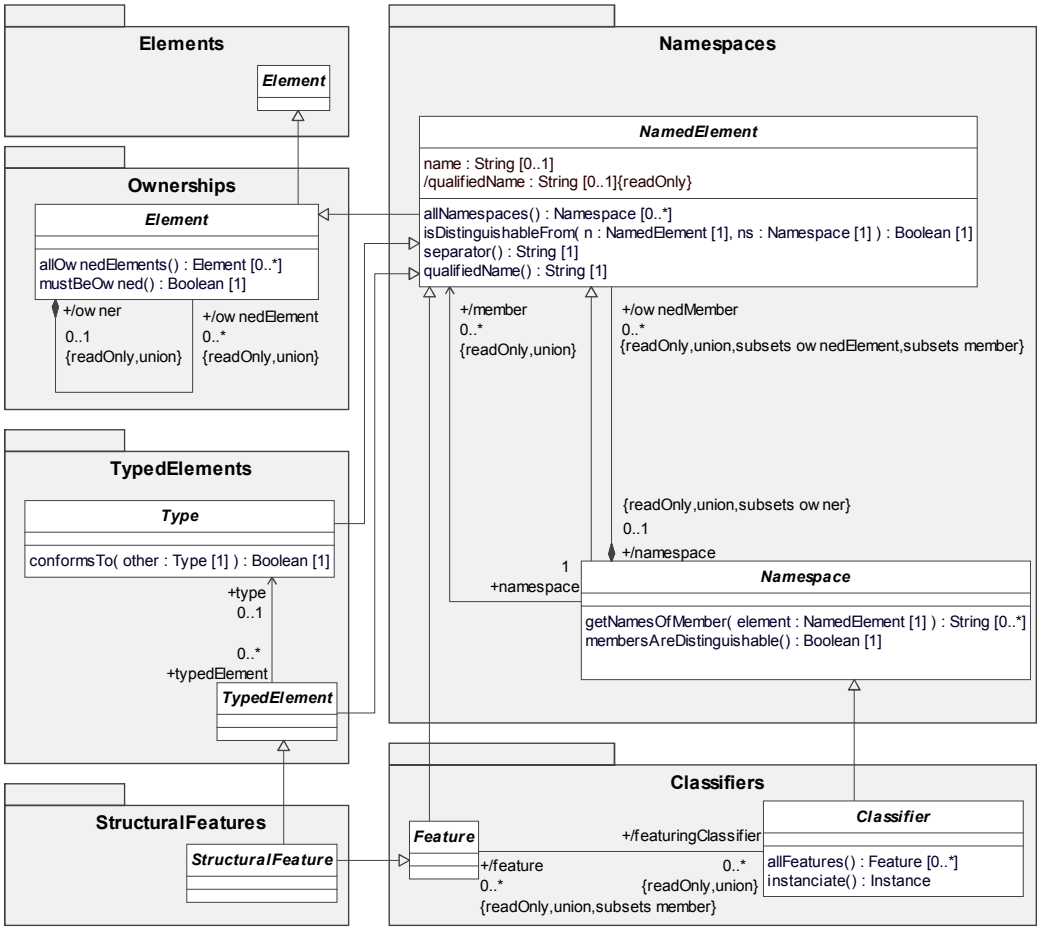


Figure 5.1: An except of the UML infrastructure abstraction library of abstract language construct definitions.

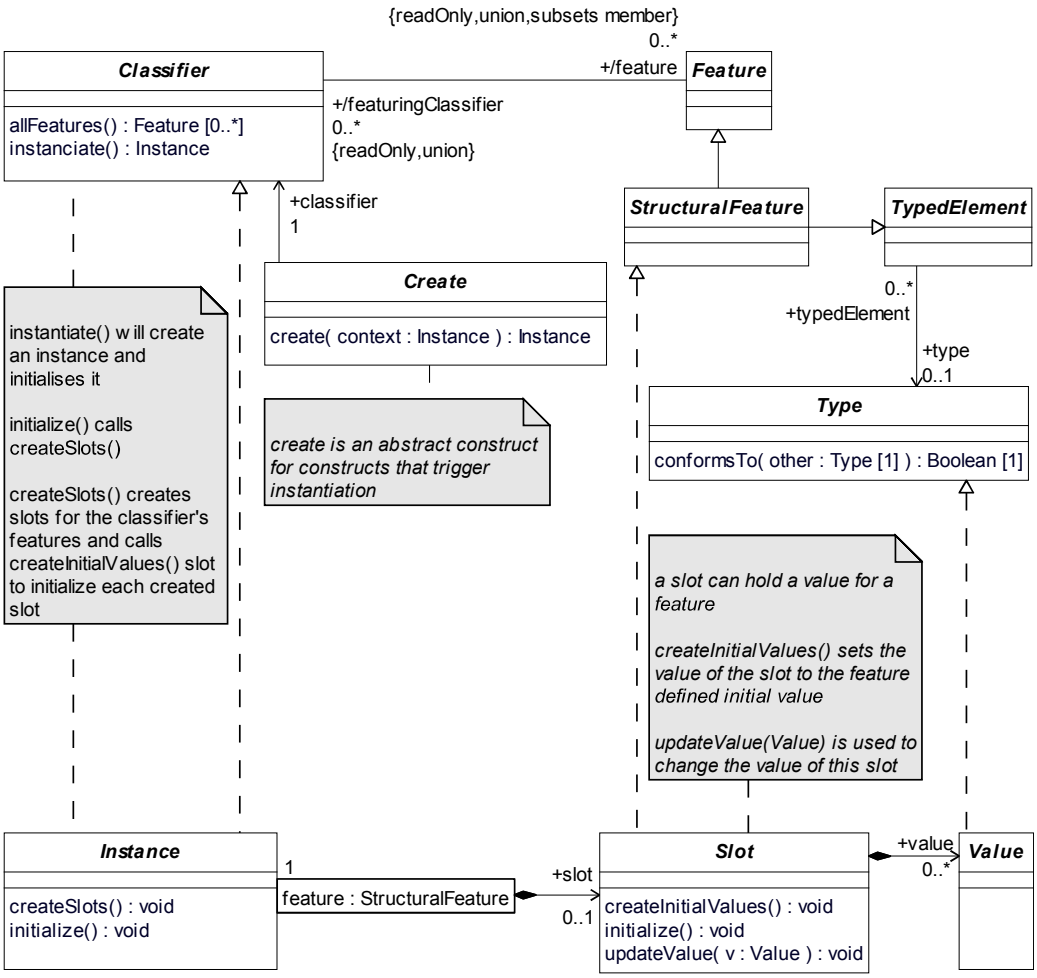


Figure 5.2: Abstract construct classes for the instantiation concept.

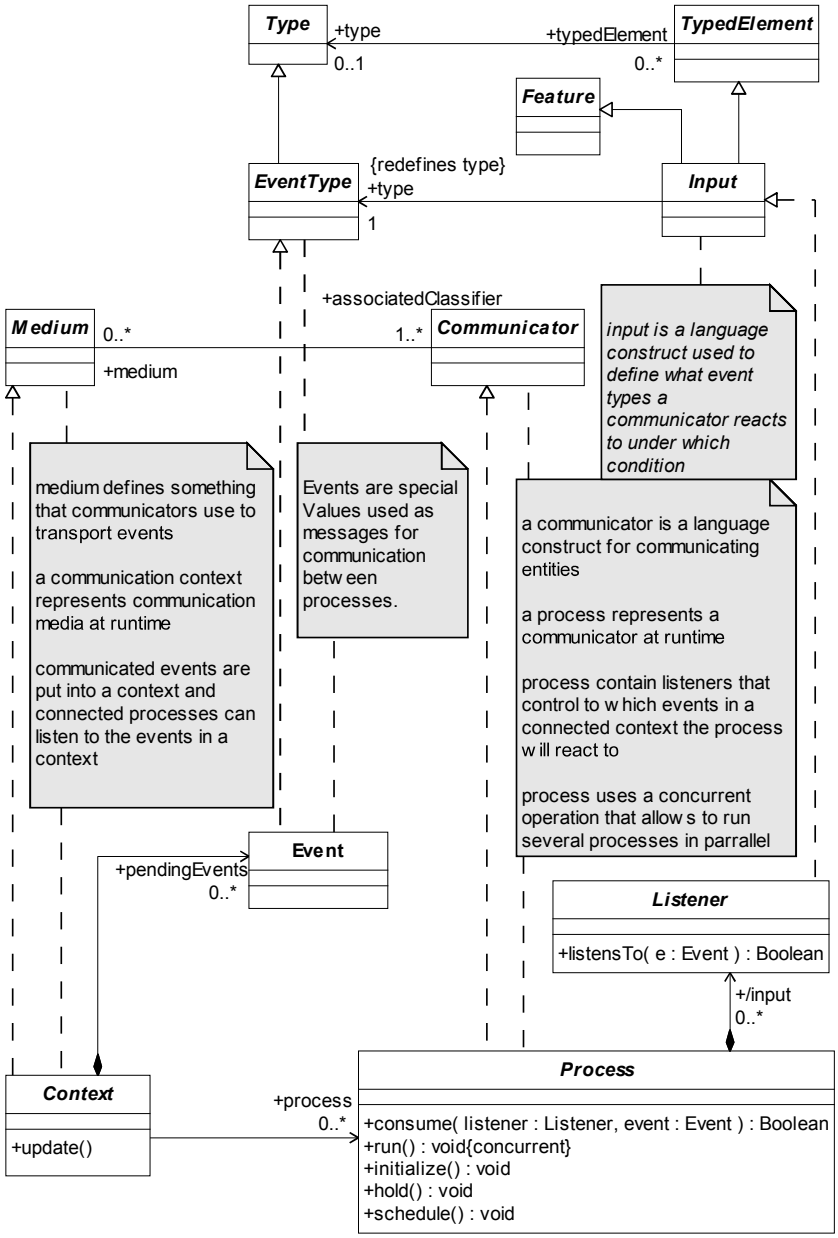


Figure 5.3: Abstract construct classes for the concurrency and communication concept.

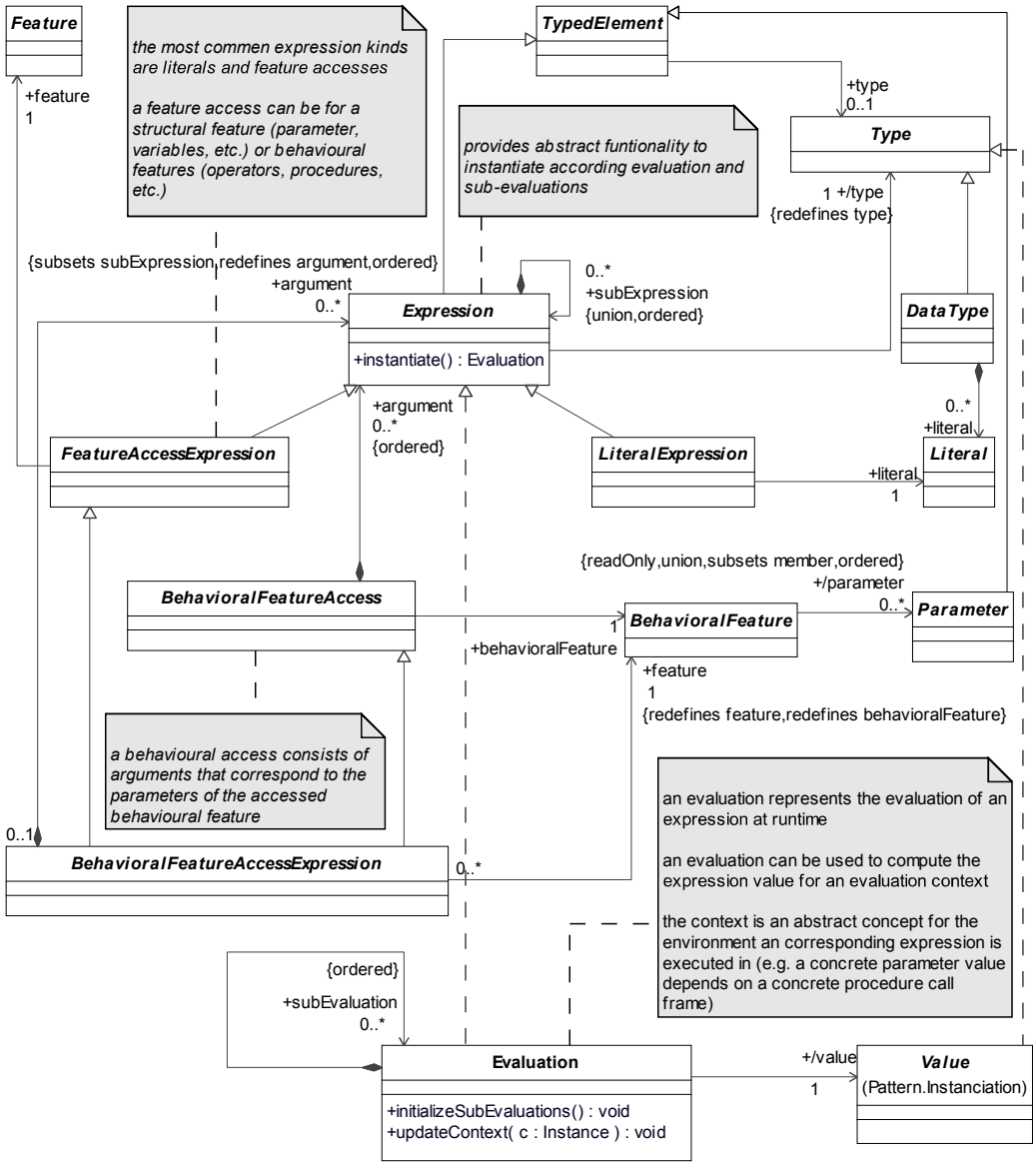


Figure 5.4: Abstract construct classes for the expression concept.

5.2.3 (Re-)Using Language Constructs

As a next step, I take the generated SDL meta-model and identify concrete occurrences of abstract constructs in the model. These concrete realisation of language concepts can now be replaced with the abstract concept realisation through sub-classing and inheritance.

Lets take the *instantiation* and *namespace* concepts as examples. The SDL-R language contains five instances of the instantiation concept: the constructs `SdlAgentType`, `SdlStateType`, `SdlProcedure`, `SdlSignal`, and `SdlTimer` can all be instantiated. In Fig. 5.5, Fig. 5.6, and Fig. 5.7 you see the different re-uses of the abstract instantiation classes (re-used classes are marked grey). All these constructs are also namespaces, in addition `SdlPackage` and `SdlDataType` form namespaces.

Further abstract concepts that are used several times are typed elements and parametrisation. Concurrency and Communication is used to model running agent instances and their signal and channel-based communication (I come back to concurrency and communication when I discuss the semantics of SDL). Expressions are used to realise expressions over SDL-based data types and their operations.

Through reusing existing construct definitions for abstract language concepts, the SDL meta-model becomes more organized. Construct classes with similar properties are sub-classes of the same abstract super-class. The language description becomes more readable and the language concepts more obvious. Utilizing object-oriented modelling for the SDL language description also positively influences the descriptions of other language aspects. Fore example, the operational semantics description (refer to section 5.4) is based on operations. Those operations can also be inherited by sub-classing abstract classes. Thus, operation behaviour can be reused as well. Basically, I transfer all advantages of object-oriented software development and modelling to language descriptions.

5.2.4 References

Removing the named referenced with actual associations is an easy task, at lease from the language aspect view point. Here the meta-model becomes even more expressive than the original grammar, because all references realised indirectly by identifiers become associations between meta-model classes. What was obscured in the grammar is now directly visible. Anyways, the matter of references becomes an issue again, when you want to edit SDL models textually. This is covered in section 5.3.

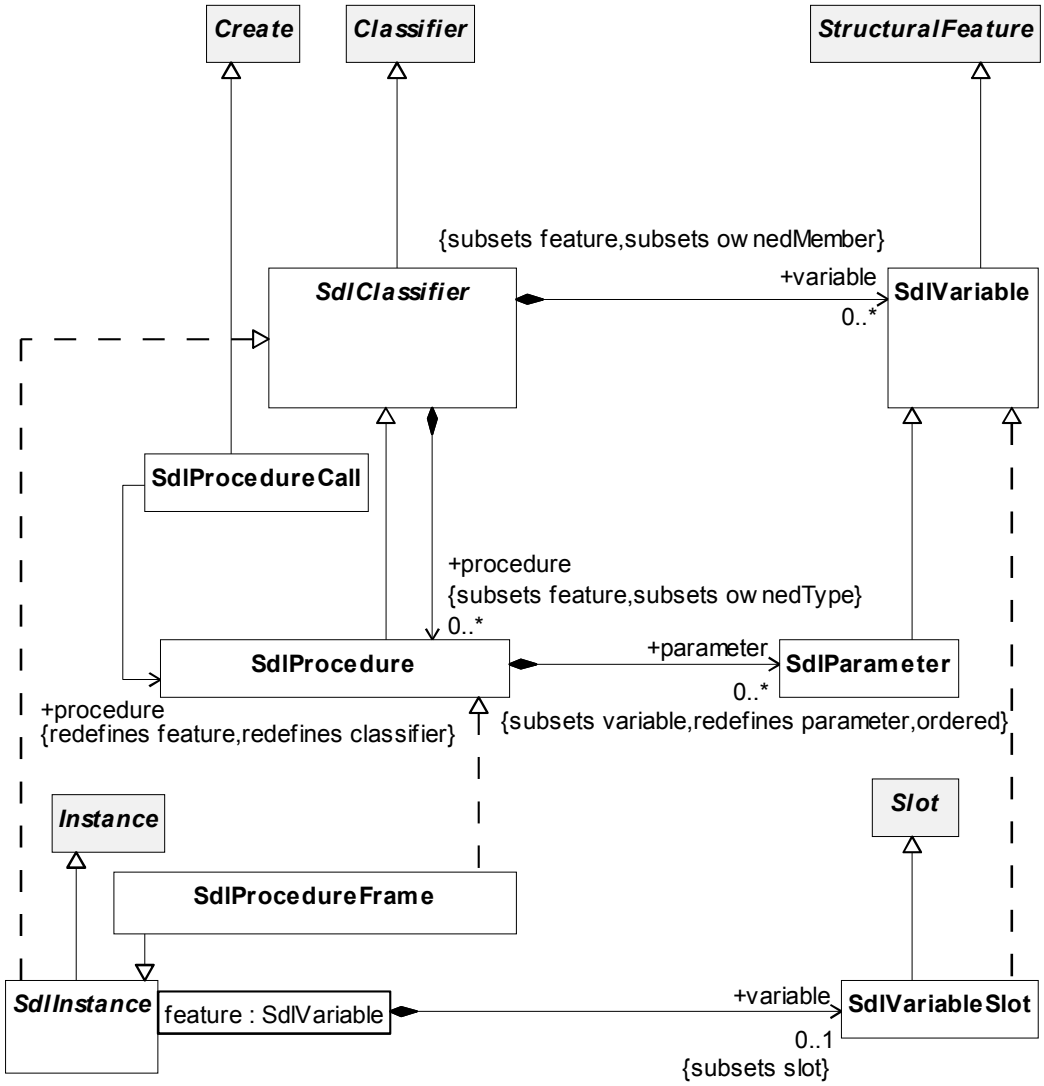


Figure 5.5: Re-use of instantiation for procedures in SDL.

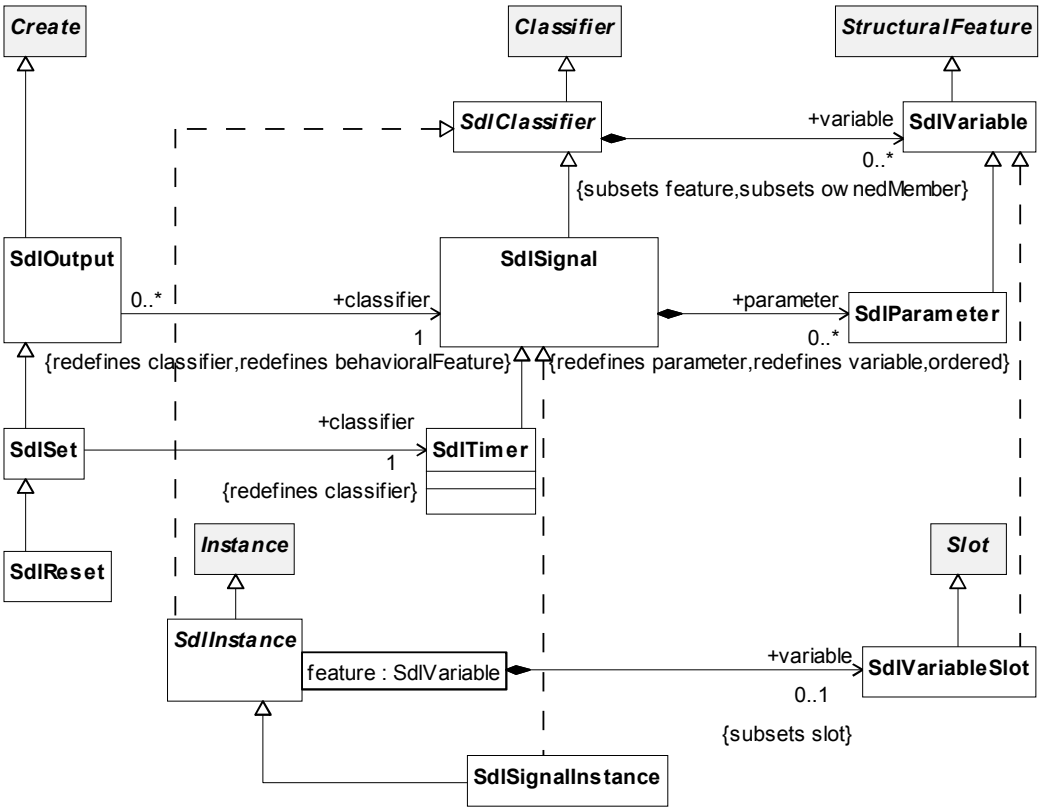


Figure 5.6: Re-use of instantiation for signals in SDL.

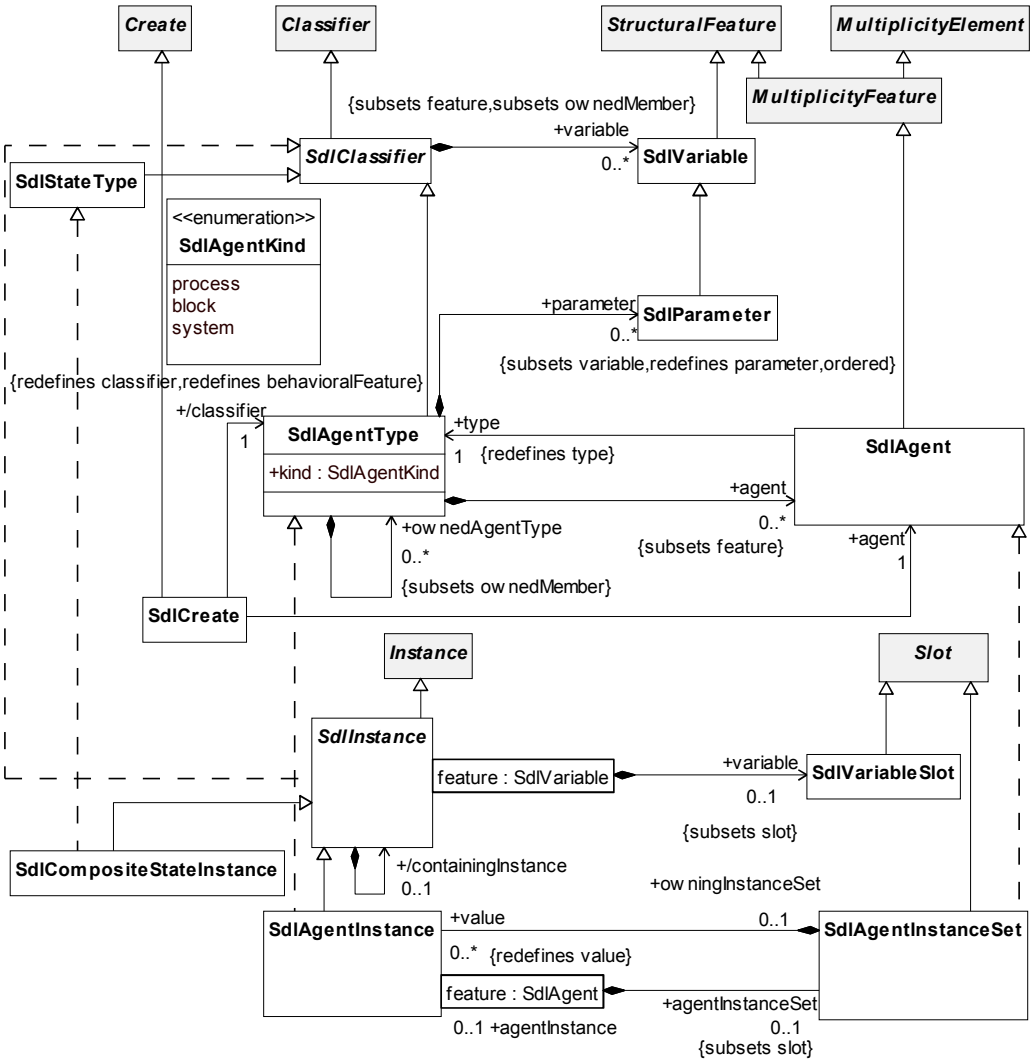


Figure 5.7: Re-use of instantiation for agents and states in SDL.

5.2.5 Structuring the Meta-Model

You can structure a meta-model into packages and into diagrams. I use packages to form encapsulated sub-meta-models that are used independently and as a cohesive whole. This means all abstract concepts, like containment, namespaces, typed elements, etc. are modelled in a different package for each abstract concept. Actually, they already are structured in this way in the UML infrastructure. I also packaged the concepts instantiation, concurency and communication, and expressions in single packages. The actual SDL meta-model is places in its own package.

I used different diagrams to logically distinguish single views on the language. I have different diagrams covering structural concepts, the different SDL namespaces, behavioural concepts, communication, expressions and data types. Diagrams don't not imply an actual meta-model structure, they are only a tool to organize the meta-model for better readability. Different diagrams often contain the same classes twice, since a class might have importance from different views on the language. The class `SdlAgentType` for example plays an important role for SDL's structural modelling capabilities, but also influences the behaviour of an SDL specification.

As a conclusion, packages allow to organize a bigger meta-model into smaller reusable sub-models. This is especially helpful for modelling libraries of abstract language constructs. Diagrams allow to increase the maintainability and readability of a big meta-model. They help to isolate certain language view points.

5.2.6 Static Semantics of SDL

Traditionally, static semantics refers to everything about a language description that cannot be covered with context-free grammars. From the MOF-like meta-modelling perspective, static semantics is a little different, because MOF-like meta-modelling is more expressive than context-free grammars (see 3.5). All issues related to identifiers and name resolutions are irrelevant for the meta-model, because these indirect references are now modelled directly with associations. Yet, you still have meta-model instances that are no valid SDL models, in a similar way that the original grammar described unwanted language instances too. The SDL standard provides semantic rules to describe constraints on language instances. These rules are provided in a first-order predicate logic calculus.

I use the *Object Constraint Language* [76] and translate the original SDL semantic rules into OCL. These OCL constraints are basically predicate logic expressions that use elements from the meta-model level and can be evaluated

```
context SDLAgent
inv: this.kind = SYSTEM implies this.container->isEmpty()
```

Figure 5.8: Example OCL constraint.

against models. The example OCL constraint in figure 5.8 expresses *An SDL agent of system kind must not be contained in another agent.* The OCL constraints themselves are part of the meta-model. They are attached to the model classes that they constrain. Evaluation of static semantic rules means: For every class all attached constraints are evaluated against every instance of this class. Since OCL constraints are attached to meta-model classes, they can be inherited as well. In other words, object-oriented meta-modelling also allows to reuse static semantics rules.

5.3 About SDL Notation

5.3.1 The Different Notations of SDL

The SDL language is standardized with two different notations. One is a purely textual notation described in a context-free grammar. The other notation, SDL's most commonly used notation, is partially graphic. This graphical notation uses wide parts of the textual notation. It uses diagrams for system structures and for the states and transitions of state machines. Everything else, data types, declarations, expressions and assignments, signatures, guards and triggers, etc. are notated textually.

In this theses, I only cover the textual notation. However, Merete Tveit presents descriptions for SDL's graphical notation in [87]. With the method presented in 3.7.5, you could combine descriptions for SDL's graphical and textual notation to create a graphical SDL editor with embedded textual editors for the respective textual model parts.

5.3.2 A Description for SDL's Textual Notation

SDL's textual notation is described with a context-free grammar in the language standard. I use the concrete SDL grammar to describe SDL's notation. Since I derived SDL's meta-model from an abstract version of this grammar, it is obvious how grammar rules and construct definitions relate to each other. Using my TEF description language is a straight forward task. I simply transfer the grammar from the grammar notation used in the standard to the grammar notation used in TEF. I reduce the grammar to those

constructs used in SDL-R. Finally, I add grammar to meta-model relations corresponding to the mapping used to derive the meta-model from the grammar. In Fig. 5.9, you see an except of SDL's textual notation description for TEF.

5.3.3 Identifiers and Name Resolution

One particular problem with describing the notation of SDL are static semantics concerns, especially names and their resolution. Within the editing process, a generated editor has to resolve named references with actual links in the model. To do this, you need to describe the SDL specific rules for identifiers (that is the name of SDL's named reference concept) and their resolution.

SDL distinguishes between simple identifiers that only consist of a name, and full qualified identifiers. Simple identifiers allow to reference elements that are known in the context of the reference. Full qualified identifiers include a path to the context that the referenced element is defined in. As many complex languages, SDL uses nested namespaces to define separable contexts that named elements can be defined in. SDL allows to use names of an outer namespace within a nested namespace but not vice versa. Furthermore, SDL uses a package import mechanism that imports the names of a different package into the actually used package.

As a default reference resolution, TEF only provides a very primitive mechanism that identifies all elements with a name attribute by their name. This happens for the whole model: there are no namespaces and no form of qualified identification. But the framework allows you to program your own identification schema. I already pointed out that the high-level description of identification mechanisms is an unresolved issue in meta-languages for textual notations. Therefore, I don't have a better solution than to realise SDL's identification and name resolution rules using the Java programming language. To achieve this, I used classical techniques known from compiler construction.

5.3.4 More Static Semantics

I already explained how I realised constraints for the SDL meta-model in section 5.2. These constraints are not part of the notation description, but part of the meta-model. But, I want the generated SDL model editor to use these constraints to validate the edited models and show errors if constraints are violated. TEF therefore automatically, evaluates all constraints of the meta-model on the actual edited model with each editing step. The according

```
Specification:element(SdlSpecification) ->
        (Package:composite(packages))*
        MainSystem:composite(system)
        (Package:composite(packages))*;

MainSystem -> Agent;

Package:element(SdlPackage) -> "PACKAGE"
    IDENTIFIER:composite(name) ";"
    (PackageContent)*
"ENDPACKAGE" ";";

PackageContent -> Agent:composite(ownedMember);
PackageContent -> AgentType:composite(ownedType);
PackageContent -> DataType:composite(ownedType);

// Agent and Agent Types
Agent:element(SdlAgent) ->
        AgentKind IDENTIFIER:composite(name)
        (AgentMultiplicity)? ":"
        AgentTypeReference:reference(type) ";";
...

AgentMultiplicity -> "(" (INTEGER:composite(lower))? ","
    (INTEGER:composite(upper))? ")";

AgentTypeReference -> Identifier;
ValueTypeReference -> Identifier;

AgentType:element(SdlAgentType) ->
    AgentKindSystem:composite(kind) "TYPE"
        IDENTIFIER:composite(name) ";"
        (AgentTypeContents)*
    "ENDSYSTEM" "TYPE" ";";
...
AgentTypeContents -> SignalDefList;
AgentTypeContents -> Agent:composite(ownedMember);
AgentTypeContents -> AgentType:composite(ownedType);
...
```

Figure 5.9: An excerpt from the SDL notation description.

OCL constraints are defined for a specific context meta-class. A constraint is always evaluated on all the instances of this meta-class. When a constraint is violated, TEF marks the context element that the violated constraint was evaluated on with an error annotation.

5.4 About SDL Semantics

5.4.1 General Discussion

Describing SDL semantics consists of two tasks based on the method described in chapter 4. The first task is to describe all the needed runtime constructs as part of SDL's meta-model. The second task is to fill the meta-model element with behaviour.

As mentioned earlier, I used object-oriented techniques to structure the meta-model. This includes the use of reusable abstractions for certain language concepts. This also effects the description of operational semantics: runtime constructs and their behaviour can be reused as well.

5.4.2 Concurrency and Time

SDL is used to describe systems of concurrently running agents. The semantics description has to reflect this concurrency. With the operational semantics description language described so far and in examples used so far, I only described single threads of evolving system states. This means a single sequence of states. You can use my method to create semantics with concurrently running threads in several ways. Firstly, UML activities allow to fork execution with *concurrent forks* and join execution with *concurrent joins*. This can be used to describe operational semantics with concurrency, but to realise concurrency in simulated or interpreted language instances the corresponding meta-tool has to implement forks and joins appropriately. Secondly, operations can be called asynchronously or synchronously. When operations are called synchronously the called method is executed before the execution thread continuous with the calling operation. This way, you only have one sequence of states in the final behaviour. When operations are called asynchronously, the calling operation is further executed right away, and the called operation is executed in a new executional thread. This thread is ended when the execution of the called operation is finished. Of course, the called operation can further call other operations asynchronously allowing arbitrary branched behaviours. Here again, the realising meta-tool has to implement synchronous and asynchronous operation calls appropriately.

The meta-tool developed for my operational semantics description language realises the two concurrency concepts in the following ways. Activity forks and joins are only realised in *pseudo* concurrent way. Since the activities are based on Petri-Nets, a fork is realised with a transition that consumes a token from one place and produces a token in two or more different states. Each token hereby represents one thread of execution. A join on the other hand consumes tokens from two or more different states and produces only one token in a following state, thus join two threads into one. However, the meta-tool realises Petri-nets in such away, that only one transition is fired at a time. As a result, actions in two threads are not really executed in parallel, but alternatively. While the observed behaviour still seems to contain concurrently running threads, this form of *concurrency* heavily depends on the quality of the non-determinism of firing of simultaneously activated transitions. This *non-determinism* is realised very poorly. As a result, of two running threads, one is completely executed before the other is executed. This would still result in valid SDL behaviour, because the standard does not elaborate on how exactly concurrently running agents have to be executed. But, I want to assume the need for a more fair execution of agents than just giving on agent always priority over all others.

Asynchronous operation calls are implemented based on Java threads and allow a more realistic concurrent execution. Therefore, I use this concept to realise running agents in SDL in particular, and concurrency as an abstract language concept in general. This abstract language concept is modelled as a set of language and runtime construct definitions in figure 5.3. The runtime class `Process`, which models concurrently running processes has a *concurrent* run operation. Callers use this operation to start the process and thereby create a new thread of execution. The operations `hold` and `schedule` allow to manipulate the thread: they allow to hold the execution and to re-schedule the execution from a different thread. This operation use normal Java thread synchronisation concepts to work.

Time plays another integral role in SDL systems. In SDL's semantics, all transitions and actions taken in these transition don't take any time (SDL-R does not support delaying channels). This means the only point where a SDL system consumes time, is a system waiting for a timer. SDL allows to set timers and allows timers to trigger a transition in an agent at a certain point in time. This concept of time, has to be described somehow within a description of SDL's semantics.

The handling of time depends on the nature of language instance execution: do you want to interpret a SDL system or simulate it. When a system is interpreted, you want to run it in real time; timers have to reflect the actual time. In simulations, the current time is just a value held in a variable.

Should a semantics description distinguish between those two forms of time? Preferably not, the description should be the same for simulation and interpretation. The difference is only made by the meta-tool that consumes the semantics description. This requires to make time a concept of the semantics description language, so that meta-tools can realise this meta-language concept differently, independent from the concrete semantics description.

Unfortunately, my semantics description language does not have a concept of time. Should it have one? Maybe. As with each other concept that might be worth adding to the meta-language, this is a trade off. There are probably a lot of concepts that could be added to the meta-language. But would it make it a better language? With each concept the language becomes more expressive, but, yet, not more flexible to use. Especially with time, which is realised in so many different ways and comes in so many flavours that it is hard to find one reasonable abstraction that makes a good meta-language concept. In the end, the meta-language is to be used for all sorts of language semantics and a time concept suitable for SDL might be totally inappropriate for other languages.

In the end however, my meta-language does not support a time concept and you need to look for an alternative to realise time in the SDL operational semantics. You can add operation implementations written in Java to your semantics description. Java allows you to access system time and realise a time concept based on the *real* time as it is perceived by the underlying computer system. While this is a practical approach to describe operational semantics suitable for interpretation, it makes the description depended on the execution kind, and you can't use the description to realise a simulation of SDL systems anymore.

5.4.3 Behaviour Descriptions, Java VS. UML Activities

In general, after I modelled language and runtime constructs you can decide for each operation if you want to realise it with UML activities or in the Java language. While UML activities promise a higher level of abstraction and therefore more expressive shorter operation realisations, writing them efficiently is hard based on the given tool support. I have created eclipse plug-ins to edited UML activities and connect them to the operations in the meta-model, yet this tool support has some weaknesses compared to the support for writing Java code.

For Java, you have tools that were developed and enhanced for more than a decade now. And even though Java is a multi-purpose programming lan-

guage that only allows programming on a lower level of abstraction, its tool support compensates for this disadvantage. Java tools for example allow to see static errors right on entering the code and it provides code-completion and smart navigation. Editing an OCL constraint in an UML activity diagram on the other hand can be very tedious without such features.

The experience with describing SDL's operational semantics showed that it is indeed easier and faster to describe the semantics in Java. Does this provide the better operational semantics? Hardly, the Java code is more platform depended, harder to comprehend, and it is usually more code to write. But on the other hand, it is faster during interpretation, and easier to debug.

Next generations of tools for a semantics description language similar to this presented here (e.g. [112]) address most of these issues. In the future there will be editors with error annotations and code completion (e.g. based on the techniques presented in section 3.7.5). There is ongoing work on debugging semantics descriptions based on UML activities, and finally interpreters for these kinda language descriptions become more efficient and compensate for Java's performance advantage.

In this case-study, I created an UML activity-based realisation for the more abstract operations that carry most of the high-level language semantics and implemented operations that tackle the details of SDL's semantics in Java.

5.4.4 An Example Excerpt from the Created SDL Semantics Description

Fig. 5.10 covers the part of SDL that is concerned with the behaviour of SDL agents: state automatons. In SDL, state automatons can be hierarchically composed, which means that each automaton consist of states, and each of these states itself can contain another automaton. Conclusively, also the outermost automaton is contained in an outermost state. This state represents a behaviour for an agent. As everything else in SDL, states are typed: a state automaton is defined within a state type, and used, as behaviour, via a state of that type. Each agent therefore contains a state as behaviour. On instantiation, when the agent instance is created, each agent instantiates the type of its behaviour as a composite state instance. The instantiated state type is also a classifier that can contain sub-states as features. Therefore, when the state type is instantiated its sub-states are also instantiated.

Furthermore, Fig. 5.10 shows the classes that run state automatons: `SdlAgentType` that references `SdlCompositeState` as behaviour, `SdlAgent`

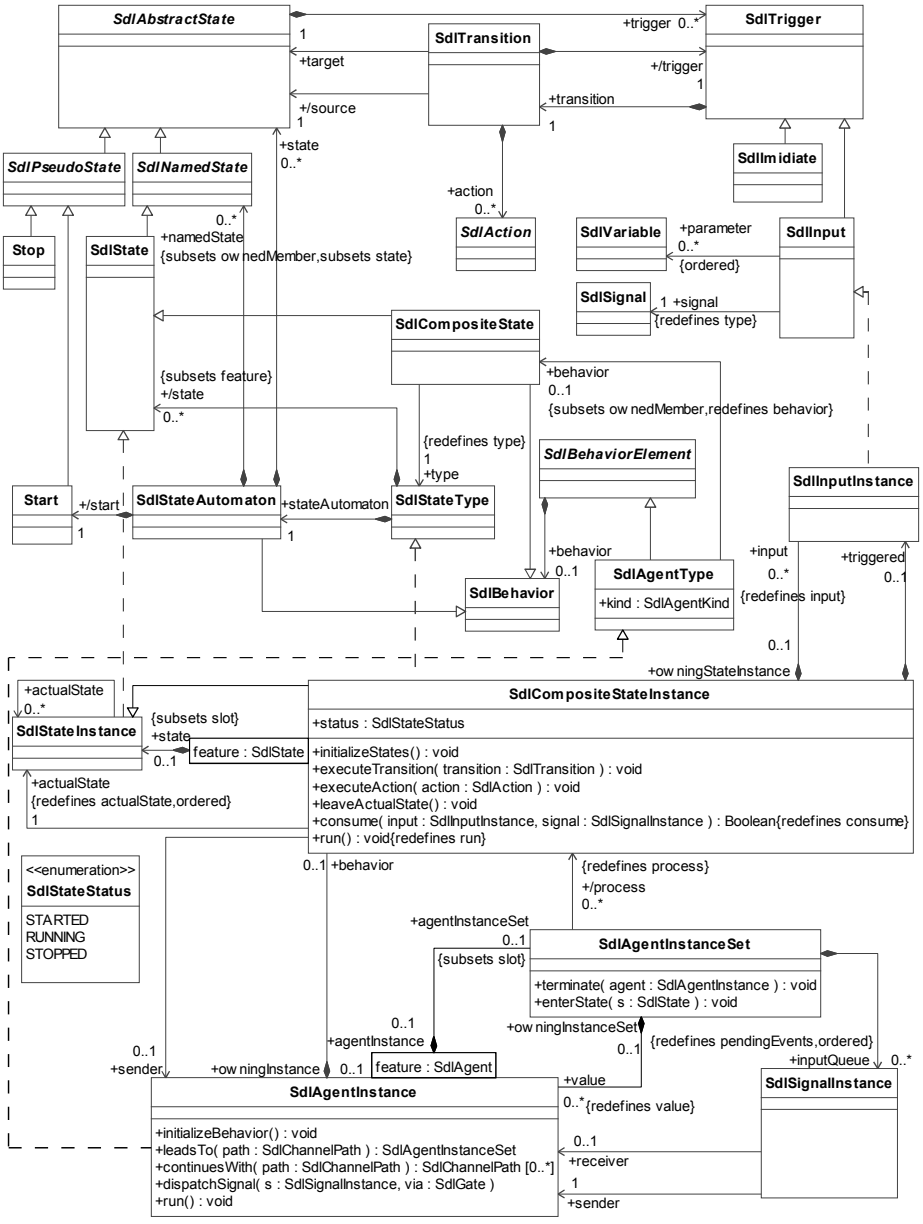


Figure 5.10: The SDL meta-model part relevant to understand state automatons and signal-based communication in SDL.

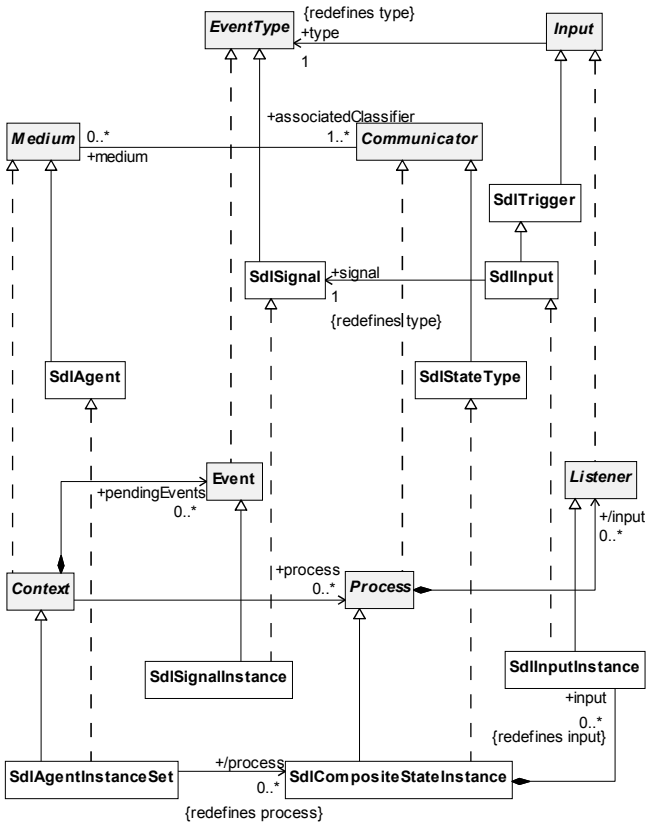


Figure 5.11: Re-use of the abstract concurrency and communication constructs in SDL.

used to syntactically represent instantiated agent types, and `SdlAgentInstance` and `SdlAgentInstanceSet` which finally represent agents at runtime.

At runtime, you finally have agent instances that contain agent instance sets for all their sub-agents. Each agent instance set contains a set of agent instances. Each agent instance contains a composite state instance as its behaviour runtime representation. A composite state instance contains other state instances as sub-states. This might either be normal state instance or composite state instances too.

Fig. 5.11 shows how these constructs extend the abstract constructs of the concurrency and communication concept shown in Fig. 5.3. You see that composite state instances are processes that communicate with signals as events. Agents and agent instance sets form communication media and contexts. Signals that are ought to be send from one agent instance to a target agent can be add to the pending events of the agent instance set that is a runtime representation of the target agent. This instance set, acting

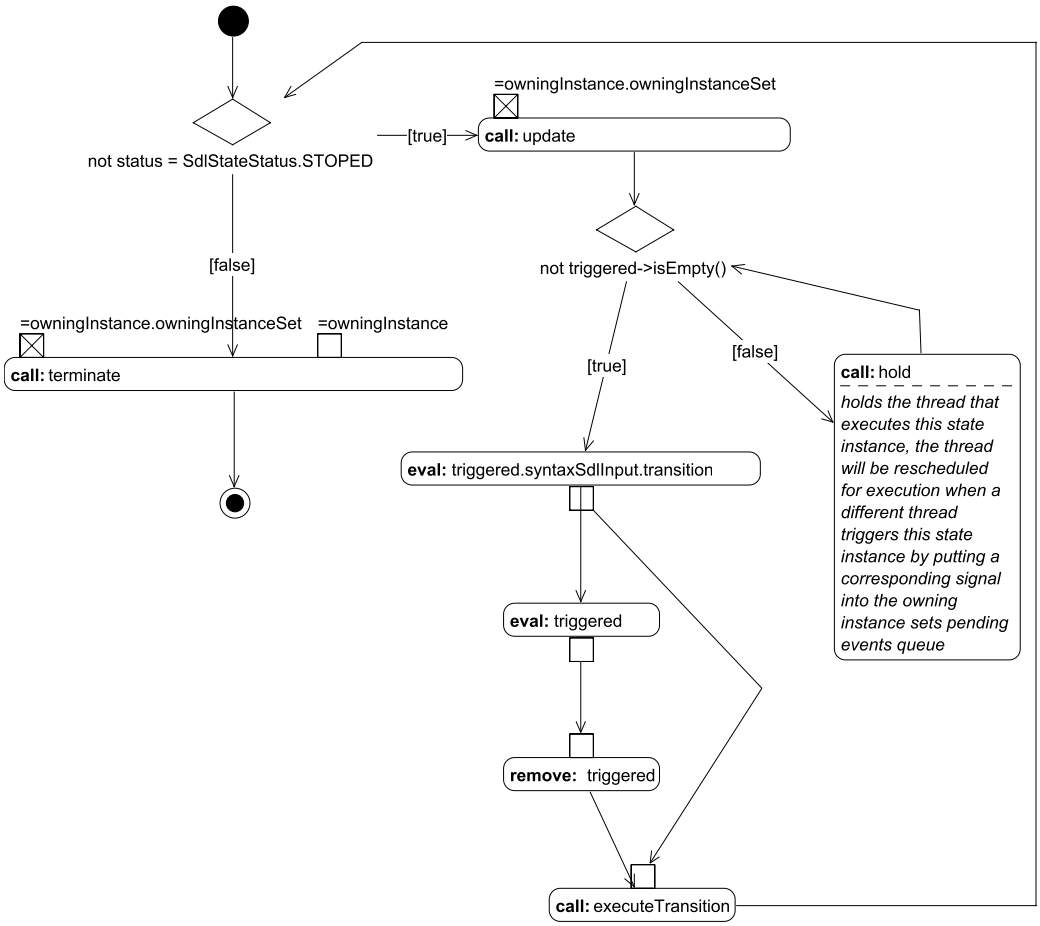


Figure 5.12: Implementation of the `SdlCompositeStateInstance::run()` operation.

as communication context, will try to make the contained composite state instances, contained in the contained agent instances, consume the signals.

In the following, I look at some of the operation implementations to discuss how SDL state automatons and inter agent communication via signals works and is realised in my semantics description language. For simplicity, I describe an SDL semantics that is even more reduced than the one of SDL-R. I am now only concerned with simple state automatons that only use inputs as transition triggers, no fancy immediate, now, save, or other triggers. There are also no complex pseudo states, such as state nodes that reference several or all states at once, etc.

`SdlCompositeStateInstance::run`

The first operation I want to look at is `SdlCompositeStateInstance::run()`. I assume that the composite state instance is property initialised; the corresponding state automaton is in a normal state; Now the `run` operation, implemented in Fig. 5.12, does the following. If the state instance has not been stopped (by reaching the stop state), it calls `update` on the surrounding communication context (the owning agent instance set). This will cause the surrounding context to try make all its processes, which contain this composite state instance, to consume signals from the pending events queue. This could trigger a transition in this state instance. Now, if a transition is triggered, it simply removes the trigger and executes the triggered transition based on the trigger. After that, everything happens from the start all over again. If no transition is triggered, this means there is no consumable signal, it simply holds the execution of the thread this state instance runs in. It hopes for another composite state instance to output an signal and to put it into the communication context of this instance. Later you see how this can cause the triggering of a transition and a continuation of this executional thread.

`SdlCompositeStateInstance::executeTransition`

Now this operation has to perform the transition. Fig. 5.13 shows the implementation. Firstly, it removes all inputs. All transition triggers are removed from the state instance, and the state instance wont react to any new signals now. Then, it executes all the action in this transition. You later see what here happens. Finally, the operation has to distinct between the kind of state that the transition leads to. If the automaton shall reach the stop state, the status is set to `STOPPED`, which will cause `run` to terminate the owning agent instance at the next iteration. If the automaton shall reach another state, this state is been made the actual state and all the triggers for this state are installed, so that the composite state instance is now reacting to all signals that correspond to inputs of transitions originating in the new state.

5.4.5 `SdlCompositeStateInstance::executeAction`

This operation executes a given action. There are several kinds of actions in SDL. Here, I only look at the output action, which is important to understand SDL's signal-based communication. An output causes the composite state instance to send a new signal from the owning agent instance to an agent instance in a possibly different agent instance set. The receiving agent

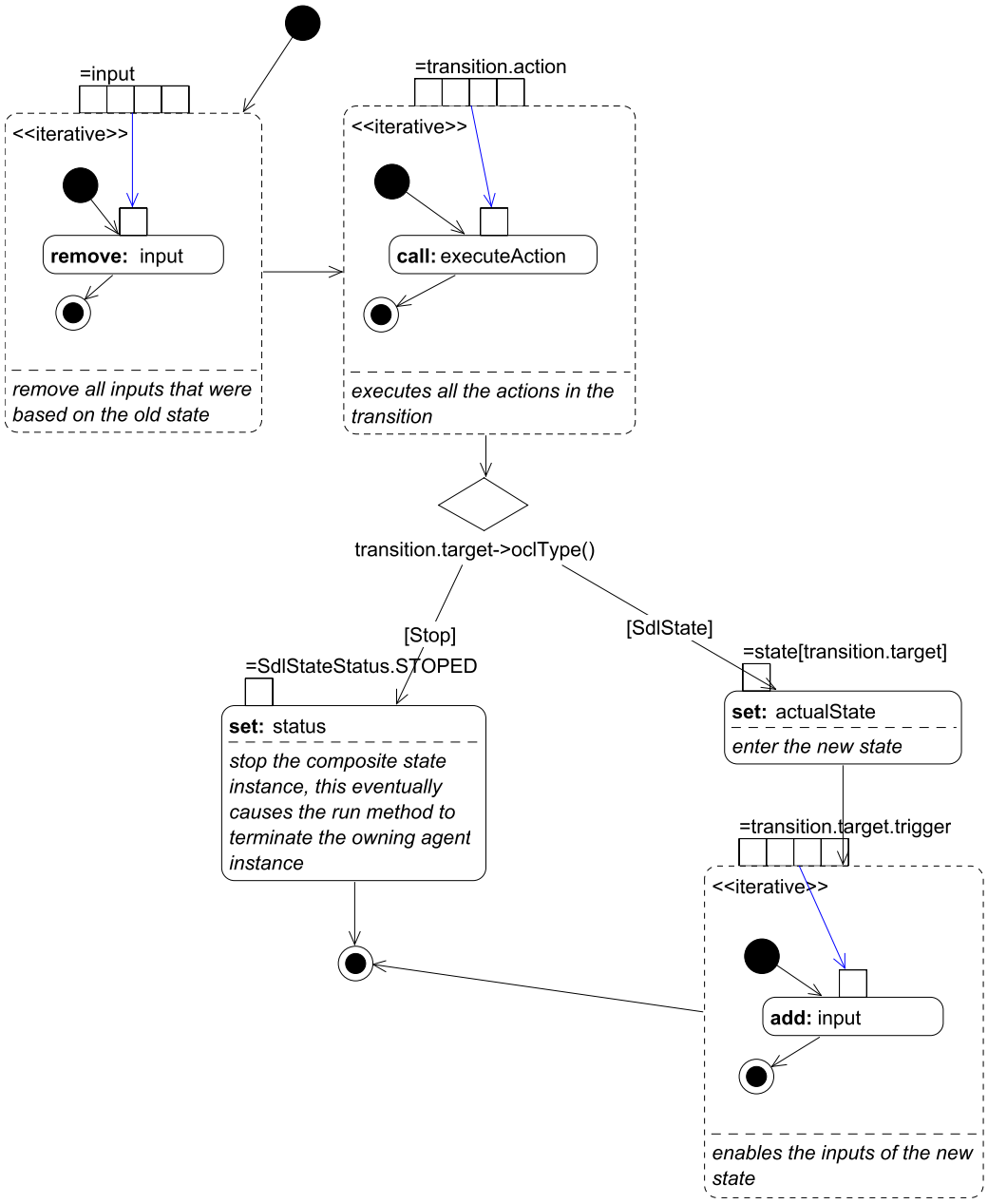


Figure 5.13: Implementation of the `SdlCompositeStateInstance::executeTransition(SdlTransition transition)` operation.

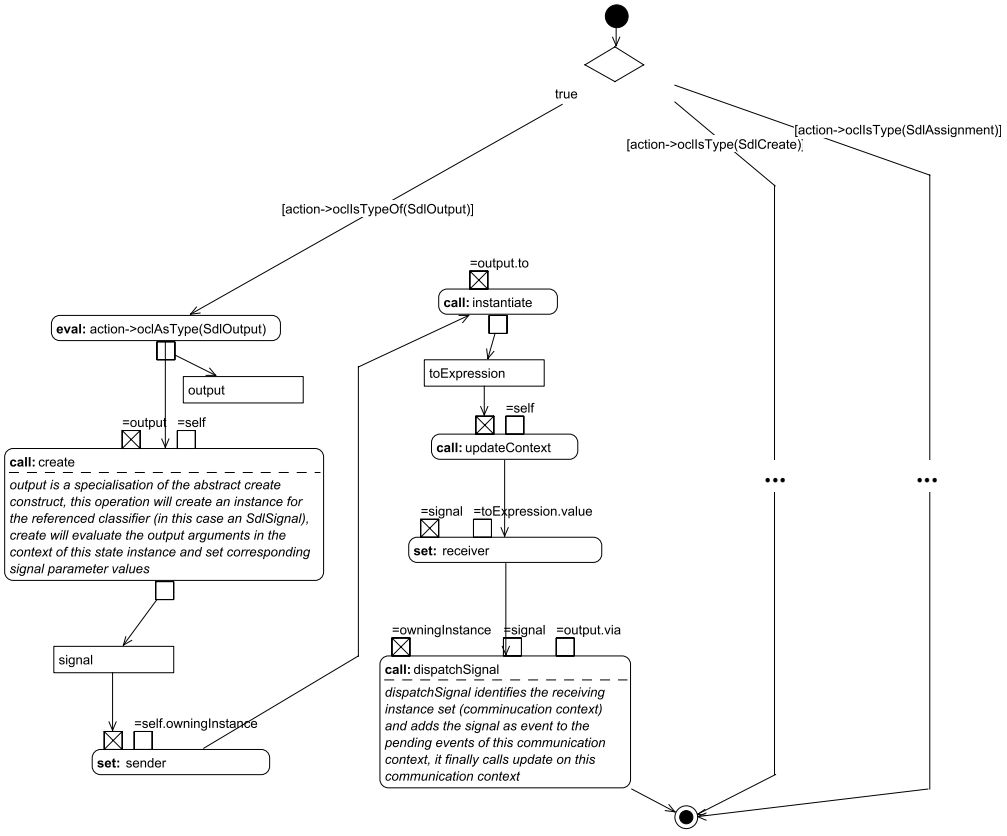


Figure 5.14: Implementation of the `SdlCompositeStateInstance::executeAction(SdlAction action)` operation.

instance set is given by the SDL specification's gates, channels, and paths, which determine the possible ways a signal can go through a specified system. Signals can be send to specific agent instances, identified by an instance id called *PID* (process ID), or without naming a specific agent instance. When no agent instance is given, the signal can be consumed by any of the agent instance in the receiving agent instance set.

The operation implementation in Fig. 5.14 also shows the behaviour for other actions. The output action is handled in the left most branch. Firstly, a new signal is created. The used operation also evaluates all argument expressions and sets the corresponding signal parameters. Then, sender and receiver are set. Sender is simply the owning agent instance. The receiver is given as an expression. Firstly, the expression is evaluated and then the signal's `receiver` property is set to the evaluation result. Finally, the signal is send, using the `dispatchSignal` operation, which is explained next.

5.4.6 `SdlAgentInstance::dispatchSignal`

This operation is rather algorithmic, computing possible agent instance set from given arguments. This is a good example for an operation, which is easier to implement in Java, because of its computational nature. Fig. 5.15 shows the shorted version of the Java implementation of the `dispatchSignal` operation. The details of this operation are rather uninteresting, it simply follows all path and channels to find possible receiving agent instance sets. After that, it goes through the identified agent instance sets, looking for the receiver agent instance. To actually send the signal, the operation simply puts the signal into the pending events queue of the receiving agent instance set (communication context) and calls the update operation on this agent instance set. What this operation does is explained next.

5.4.7 `Context::update`

The `update` operation checks for the possible consumption of the topmost pending event by one the processes in this communication context. The implementation is shown in Fig. 5.16. The operation removes the topmost event from the queue. It iterates through all processes (in my case SDL composite state instances) and all the listeners of these processes (in my case SDL inputs) that listen to the event. If one of these listening processes can actually consume the event, update is finish. If non of these listening processes can consume the event, it saved at the back of the pending event queue for later consumption.

```
public void dispatchSignal(SdlSignalInstance s, SdlGate via) {
   SdlAgentInstanceSet dispatchedTo = null;
   SdlSignal signal = s.getMetaClassifierSdlSignal();
   ReflectiveCollection<?
         extends SdlAgentInstanceSet> possibleReceiver =
         collectAgentInstanceSets(s, via, self);
   loop:
   for (SdlAgentInstanceSet receiver: possibleReceiver) {
      if (s.getReceiver() != null) {
         // the signal is dispatched to the first instance set
         // that contains the preset signal's receiver
         if (receiver.getValue().contains(s.getReceiver())) {
            dispatchedTo = receiver;
            // the signal is added to pending events of the
            // corresponding instance set (communic. context
            receiver.getInputQueue().add(s);
            // this provokes the communication context to try
            // consuming the signal in all contained processes
            receiver.update();
            break loop;
         }
      } else {
         // the signal as no receiver set: the signal
         // is dispatched to an arbitrary (the first)
         // possible receiving instance set
         dispatchedTo = receiver;
         receiver.getInputQueue().add(s);
         receiver.update();
         break loop;
      }
   }
   if (dispatchedTo == null) {
      // dispatch to environment
      ...
   }
}
```

Figure 5.15: `SdlAgentInstance::dispatchSignal` as Java implementation.

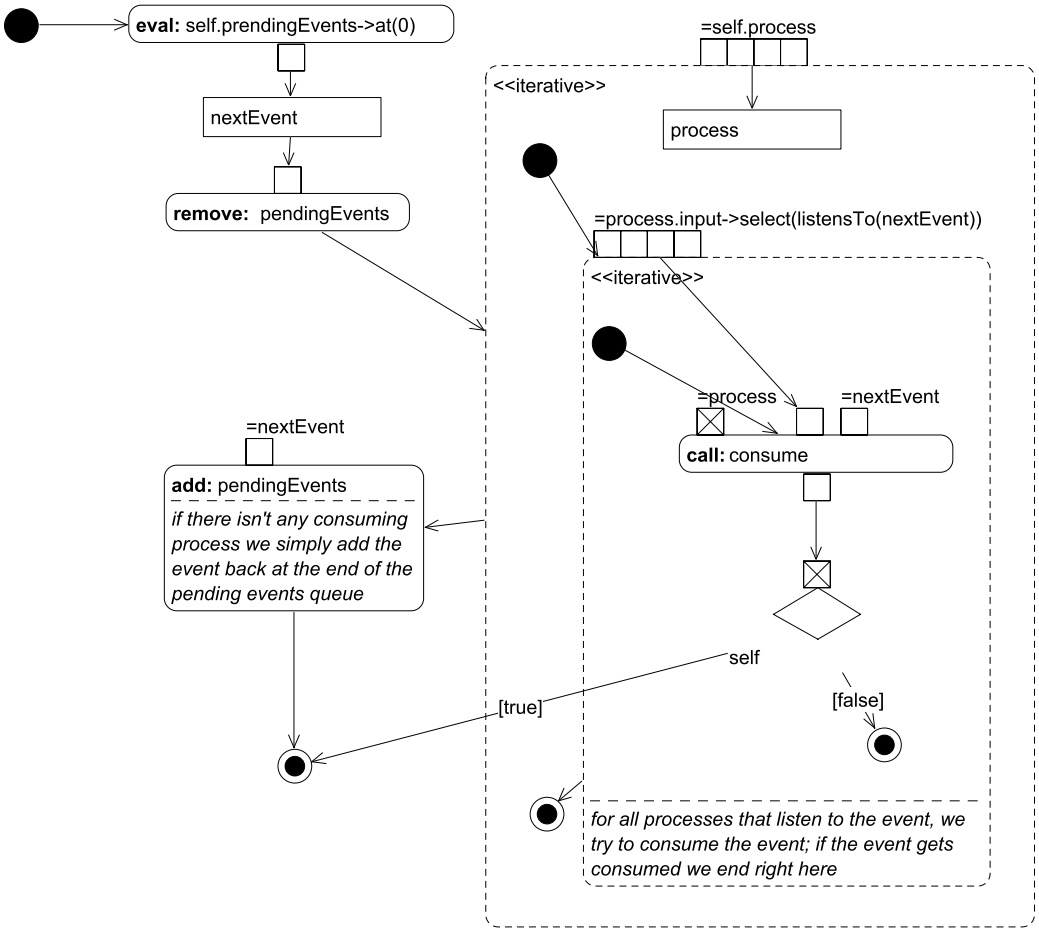


Figure 5.16: Implementation of the `Context::update()` operation.

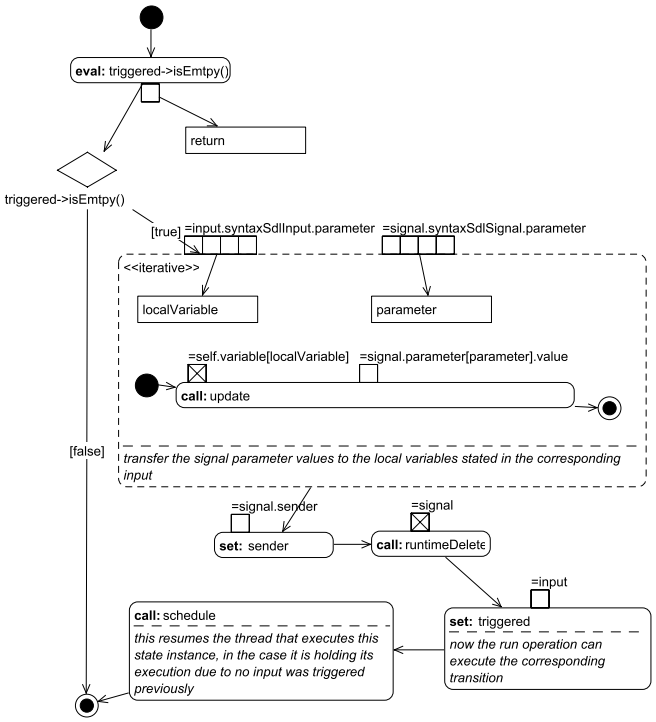


Figure 5.17: Implementation of the `Context::update()` operation.

5.4.8 `SdlCompositeStateInstance::consume`

With this operation a composite state instance tries to consume a signal. The implementation in show in Fig. 5.17. If the state instance already has a triggered transition, it cannot consume another signal and `consume` simply returns false. If the state instance does not have a triggered transition, the signal is consumed. This means `consume` takes all the signal parameter values and puts them into the corresponding local variables; it transfers the sender from the signal to the state instance; it finally deletes the signal. Afterwards it triggers the according transition and schedules the state instance for execution, becaus the state instance might be holding execution waiting for a signal to consume (see the implementation of `SdlCompositeStateInstance::run`. Here the circle clothes: the SDL state automaton logic continues with the `SdlCompositeStateInstance::run` operation that can now execute the triggered transition.

5.5 Conclusions

5.5.1 Description Efficiency vs. Tool Performance

What is won over creating a SDL tool chain with traditional means. Grammars are already given as a description for notation and language. How can the approach taken in this case-study be compared to a traditional software engineering approach to build SDL language tools? A formal comparison would not make much sense: there is no other textual editor for SDL specifications, and manually written interpreters, simulators, and compilers naturally outperform my generic operational semantics interpreter. In the end, it comes down to weigh advantages and disadvantages, since DSM (as each other rise of abstraction was before) is a trade-off between development efficiency and software performance.

Describing the textual notation and generating a textual editor from this description is definitely a success, because the textual notation description almost only consists of the given grammar. Almost effortlessly, I was able to create a valuable editing tool that did not exist before.

With the operational semantics descriptions things are a little different. With my current meta-tool the performance, even without formal measurements, has to be labelled mediocre compared to hand crafted tools (e.g. the SITE-tools [17]). Furthermore, any comparison is very vague because the functional range of the SDL covered by existing tools is very different from the SDL-R that is described. But even by eyes measure, it is obvious that describing the semantics with my method is by magnitudes more efficient. It was a single developer within a few weeks doing work worth of several man-month manual compiler implementation.

5.5.2 Description Quality

You can compare the descriptions created during this case-study to the SDL standard document on one hand and to implementation code or software models of existing SDL tools on the other. Compared to the SDL standard, my descriptions say more. They provide formal descriptions of language, notation, and semantics. My descriptions are better structured and their object-oriented design provide an understanding for the abstractions within the language. My descriptions are widely graphical and therefore provide a good intuition about the language and runtime constructs right away. My descriptions can be read and explored through UML tools and my own modelling tools and have not to be read page by page as in a standard.

Compared to a manual implementation of SDL language tools, my de-

scriptions are more comprehensible and on a higher level of abstraction. To some extent, my descriptions can be seen as a software model for SDL language tools. Unfortunately, there are no such models that I know of that I could compare to my descriptions.

Since, the descriptions are object-oriented models they are reusable and maintainable in a similar way. They can be refactored (for now without tool support), changed, and advanced based on object-oriented development techniques.

5.5.3 Tool Quality

In this case-study, I described the textual notation and operational semantics of SDL. Because I used meta-languages that I have meta-tools for, I can generate SDL tools from these descriptions: I generate a textual model editor from the notation description and a SDL interpreter from the operational semantics description.

The textual model editor is the first SDL editor of its kind. All existing SDL editors are graphical editors. Sill I can partially compare those editors, because graphical editors have to support textual editing for many SDL constructs for which only a textual notation exist. And here I achieved an advantage over the existing manually crafted editors, which only allow simple string-based text editing capabilities without syntax-highlighting, code-completion or instant error annotations, not to mention smart navigation. Example industry used SDL editors are Cinderella [16] or PragmaDev Real Time Developer Studio [86]. My DSM developed SDL text editor could be almost fully generated from the notation description. Only SDL's identification scheme had to be implemented as Java code (refer to section 5.3).

The SDL interpreter on the other hand cannot stand a serious comparison with manually built existing SDL tools, at least the fully automatically generated version that I discuss here. Interpreting UML activities is just not fast enough. But this might just be a meta-tooling issue, hence following operational semantics description meta-languages and meta-tools show far better performance results. The M3Actions [112] project in particular achieves this because it is based on the more light-weight eclipse EMF meta-modelling platform and uses a considerably more modern and faster OCL implementation. Even, I have to qualify my SDL interpreter as prototypical, it is still a valuable tool, because the DSM approach allows me to alternate SDL semantics quickly. You can use my semantics description and generated SDL tool as an experimental platform to evaluate different semantics variation for frequently discussed SDL constructs.

Chapter 6

Conclusions

I already concluded each chapter for itself. Here I want to look into the overall conclusions of this thesis: what are the contributions made, is the aim of the thesis fulfilled, its hypothesis proved, what is the impact and what future work. After answering these question, I end the thesis with a brief summary.

6.1 Contributions

The following is a list of the new original scientific contributions made in this thesis and a list of the peer reviewed conferences and workshops, used to publish these.[1]

- I created a mechanism to realise MOF 2.x property sub-settings and redefinitions in dynamic programming environments and implemented it in a meta-modelling framework called *A MOF 2 for Java*. I furthermore developed a object-oriented programming language mapping for MOF, which facilitates generics and co-variant return types. This mapping is also implemented in *A MOF 2 for Java*. [102]

- I created a formal model for the relationship between context-free grammars and MOF meta-models ([31]). I used this model to create a meta-language for the description of textual notations and used this meta-language to derive eclipse text editors with semantic rich editing capabilities. This work allows for the first time to describe content-assist [106] and the integration of textual and graphical editors [107].

[1] This list is a more detailed refinement of the contributions given in the introduction (1.4).

- I developed a method to use MOF meta-models as configurations for a Plotkin-like description of operational semantics. Therefore, I introduced a separation between language and runtime classes in MOF meta-models and combined UML activities with OCL into an action language for MOF. [108, 28]
- I developed a transformation language as an extension to the Java programming languages (not included in this thesis). This language allows a very simple and pragmatic development of all sorts of transformations, independent of the used meta-modelling language. [103]
- I created a meta-model, static analysis description, textual notation, and operational semantics for SDL. This allowed to create an SDL text editor including static analysis and interpreter automatically.

6.2 Results: Aim and Hypothesis

Before I conclude whether or not I achieved the goal of this thesis, you should recall the original aim and hypothesis of this work.

> *In this thesis I wanted to apply OOMM-based techniques to all important aspects of language descriptions and thereby reason about how well suited OOMM-based techniques are to describe modern computer languages.* To do this I wanted to prove the following hypothesis: *all aspects of the SDL language can be described with specialised meta-languages and the resulting descriptions of SDL allow to generate a complete prototypical tool-chain for SDL.*

Firstly, I introduced all techniques, meta-languages, and meta-tools that are necessary to describe all important language aspects (language, notation, semantics) and are necessary to DSM the corresponding language tools (editors and simulators or interpreters). Secondly, I applied these techniques to the SDL language.

In the last section, the section that discussed a case-study about applying my language description and DSM techniques to SDL, I concluded that you can not reach the overall tool quality of manually crafted tools. But, you also saw that the DSM created tools can function as valuable prototypes and that DSM of language tools is in principle possible. Generally, I can conclude that with existing meta-modelling description languages and meta-tools as well as languages and tools developed as result of this thesis work, you can describe all important language aspects, and you can automatically generate prototypical language tools from these descriptions. The hypothesis fulfilled.

I want to take a closer look at how well the aim of the thesis was reached. Although, the hypothesis is already proved, lets take a closer loot at the requirements for modern computer language descriptions listed in the introduction (1.2.4), and lets discuss how well OOMM and DSM-based on OOMM fit these requirements.

Language instance and representation have to be two separate things. MOF-like meta-modelling formalisms within OOMM do only allow to describe abstract language instances. Separate descriptions of notations are mandatory. There always is a separation between representation and language instances. That this characteristic is important became evident in my work on combining textual and graphical editors in 3.7.5.

Language description techniques must provide the means to express abstractions within language constructs. OOMM, especially with the newly created platform *A MOF 2 for Java* in 2.5, allows to express abstractions within language constructs. In the SDL-case study, I used this to reuse language constructs from the UML infrastructure and to write new abstract language constructs used within the SDL descriptions. You also saw, that these abstractions led to reuse within the operational semantics description of SDL.

Language descriptions must allow the efficient development of language tools and prototypes. OOMM-based DSM, how it works, how it can be used, and what its limitations are, was thoroughly discussed throughout this thesis. The SDL-case study showed that it can be used to create prototypical tools efficiently. But it also showed that for tool quality, achieved through manual tool implementation, the automatically developed tools need to be manually enhanced through common programming based development. At this point, it is not clear whether this in an inherent problem of DSM, or whether further enhancement of meta-languages and meta-tools can overcome this limitation. Nevertheless, automated language tool development provides a good starting point in developing language tools more efficiently.

Language descriptions must be well aligned with predominant programming paradigms to support tool development. As it become evident in the last point, it is unreasonable to assume that industrial scale language tools of high quality can be developed fully automated. The OOMM techniques presented here all allowed to further enhance the automatically developed tools manually based on programming in a multi-purpose programming language (Java). Since OOMM descriptions seamlessly map towards the concepts of object-oriented programming languages (2.5), programmatic

extensions of language tools feels natural.

It must be possible to combine different language descriptions. This point was not addressed here. But, reusing parts of the UML language description (meta-model) for the SDL case-study showed that combining different language descriptions based on object-orientation concepts like specialisation could be possible.

Despite some unresolved issues, I generally showed that OOMM-based techniques are well suited to describe modern computer languages and that I have reached the aim defined at the beginning of my work.

6.3 Impact and Future Work

This thesis was written when the growing popularity of the eclipse platform in combination with MOF-like meta-modelling in the form of EMF [117] gave a new drive to developing domain specific languages based on DSM. The presented work influenced the community work on meta-languages, meta-tools, and language workbenches in many ways:

Besides my case-study of applying MOF-like meta-modelling techniques to SDL, DSM of languages has become the major methodology in creating domain specific languages. Over the last years many meta-languages and corresponding tools have been researched and developed that cover the language aspects presented here, as well as other aspects and sub-aspects (refer to the related work sections of the corresponding chapters). Among them are textual notations [48, 69, 59, 83, 13], graphical notations [119], different forms of semantics (e.g. transformation semantics) [23, 22, 115, 47, 92, 96, 98], and debugging [98].

Not only are there now single meta-language and meta-tool pairs available, but whole, so called, *language workbenches*: GME [21], open Architecture Ware [63], metaedit+ [121], Microsoft DSL Tools [64], Ceteva XMF [71, 12], and Safari [123]. These are sets of aligned meta-languages and meta-tools. These workbenches usually contain languages and tools for all necessary aspects of a certain kind of domain specific language. Workbenches do not only allow to describe languages and generate basic tool support, but they allow to create very coherent, well aligned tools. Such tools are usually based on the same meta-modelling platform, they share file formats, and allow to trigger actions from within each other (e.g. start an interpreter from within an editor, or move the editor focus during debugging, etc.).

During the course of this thesis, you saw that you can describe languages and automatically create language tools *in principle*. This does not mean

that everything is perfect: most of the created tools are only partially automatically developed and do not reach the usual industrial strength quality of manually crafted tools. And even though the DSL and language community in general and this thesis in particular went a long way over the last five to three years, we are still only scraping on the full potential of DSM of languages.

Conclusively, there are several interesting projects that are directly based on the ideas and technology presented in this thesis. Furthermore, there is a vast number of projects working on bettering language DSM in general. I want to present a few of those projects that directly cover the open issues discussed in this thesis.

- EProvide and ESemantics [98, 96] can be used together to describe operational semantics in a more flexible way. This technology allows to describe both simulation and interpreter semantics at once. As much semantics as possible is described once for both, simulation and interpretation, and only the parts that really make a difference are described in two separate versions [94, 93, 95]. Furthermore, these projects allow to visualise language instance execution and allow the stepwise execution for visualisation and debugging purposes. On top of that, this technology also allows to combine different meta-languages to describe state transitions and transition constraints [100]. Many popular languages like QVT, ASMs, Prolog, or Scheme are already supported, other languages can be easily integrated. This allows to choose the language that is most suitable to describe the given semantics [99].

- The project EPromote [97] emulates attributed grammars with Prolog-based model transformation for MOF-like meta-modelling within eclipse. This projects uses model transformation to parse text. The text is represented as a model consisting of sequence of symbols. Model transformation is applied to this *text model* and results in the desired language instance. The expressiveness of QVT and Prolog allows to write transformations that resemble attributed grammars and allow to describe static semantics issues (such as name resolution) as part of the transformation. This allows descriptions for textual notations that cover more than context-free syntax. This technology also allows to create *error/warning models* as a side product, which can be used to display errors and warnings within the text. All notation aspects, context-free syntax, name resolution, static semantics, and constraints, can be described in a single description.

- M3Actions [112] is based on the behaviour description language that I

used in this thesis. M3Action revised tool support and provides better and more flexible editing capabilities, integrating the editing of meta-models and behaviour descriptions. Furthermore, M3Actions realises the meta-language in a more efficient meta-tool that allows faster interpretation of language instances and thereby solves the lacking performance issue that I discovered. M3Actions also works on debugging support for the language description.

- M3Actions [112] and EProvide [98, 100] work on describing the debugging aspect of languages. The goal is to describe what debugging means for the described language. Based on this debugger description a corresponding meta-tool creates a debugger tool automatically. This allows to debug language instances simulated or interpreted by language tools created from according operational semantics descriptions.

6.4 Summary

In this thesis, I researched object-oriented meta-modelling (OOMM) and how it can be used to describe computer languages. Thereby, I not only focused on describing languages, but also on utilising the language descriptions to automatically create language tools from language descriptions. I used the notion of meta-languages and meta-tools. Meta-languages are used to describe certain language aspects (such as representation or semantics) and meta-tools are used to create language tools (such as editors or interpreters) from corresponding descriptions. This combination of describing and automated development of tools is known as domain specific modelling (DSM). I successfully applied DSM based on OOMM to describe all important aspects of executable computer languages. I chose SDL, the Specification and Description Language, as an archetype for textually notated languages with executable instances. For this archetype, I showed that the presented meta-languages and meta-tools allow to describe such computer languages and allow, in principle, to automatically create tools for those languages. But while I showed the basic applicability of OOMM and DSM to languages, this work can only mark a starting point, and there is a lot of future work to come in order to reach the desired tool quality and cover the whole diversity and complexity of today's computer languages.

Bibliography

[1] Alfred V. Aho, Ravi Ethis, and Jeffrez D. Ullmann. *Compilerbau Teil 1.* Oldenbourg, Wien, 2 edition, 1999. German translation of "Compilers, Principles, Techniques and Tools".

[2] Marcus Alanen and Ivan Porres. A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS, 2004.

[3] Marcus Alanen and Ivan Porres. Basic Operations over Models Containing Subset and Union Properties. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 2006.

[4] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *A. Rensink, J. Warmer (eds.), Model Driven Architecture - Foundations and Applications: Second European Conference*. Springer Verlag, 2006.

[5] Carsten Amelunxen, Andy Schürr, and Lutz Bichler. Codegenerierung für Assoziationen in MOF 2.0. In Thomas Kühne, editor, *Modellierung 2004*, LNI, Bonn, 2004. Gesellschaft für Informatik.

[6] Andy Schürr et. al. MOFLON. http://www.moflon.org, 2008.

[7] Rolf Bahlke and Gregor Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576, 1986.

[8] Joseph Bergin. Building Graphical User Interfaces with the MVC Pattern, 2007.

[9] E. Borger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[10] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[11] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *Int. Journal of Computer Simulation*, April 1994.

[12] Ceteva. XMF. http://www.ceteva.com/xmf.html, 2008.

[13] Philippe Charles, Julian Dolby, Robert M. Fuhrer, Stanley M. Sutton Jr., and Mandana Vaziri. SAFARI: A Meta-Tooling Framework for Generating Language-Specific IDE's. In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 722–723. ACM, 2006.

[14] N. Chomsky. Three Models for the Description of Language. *IEEE Trans. Information Theory*, 2(3):113– 124, 1956. ISSN 0018-9448.

[15] Noam Chomsky. *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA, 1965.

[16] Cinderella. Cinderella SDL. http://www.cinderella.dk/, 2008.

[17] Cinderella. SDL Integrated Tool Environment (SITE). http://www.cinderella.dk/, 2008.

[18] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 2nd edition, 1991.

[19] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Yourdon Press, Upper Saddle River, NJ, USA, 1991.

[20] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[21] James Davis. GME: The Generic Modeling Environment. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 82–83. ACM, 2003.

[22] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2002.

[23] Juan de Lara and Hans Vangheluwe. Defining Visual Notations and Their Manipulation Through Meta-Modelling and Graph Transformation. *J. Vis. Lang. Comput.*, 15(3-4):309–330, 2004.

[24] Zinovy Diskin and Jürgen Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2006.

[25] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. *onBoard*, November 2004.

[26] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-CASE in Practice: a Case for KOGGE. In Antoni Olivé and Joan Antoni Pastor, editors, *CAiSE*, volume 1250 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 1997.

[27] Hartmut Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1979.

[28] Hajo Eichler, Markus Scheidgen, and Michael Soden. A Semantic Meta-Modelling Framework with Simulation and Test Integration. In Hajo Eichler and Tom Ritter, editors, *Proceedings of the ECMDA Workshop on Integration of MOdel Driven Development and Model Driven Testing*, pages 51–60. Fraunhofer IRB Verlag, 2006.

[29] Karsten Ehrig Claudia Ermel. Modeling Visual Languages based on Graph Transformation Concepts and Tools. In Felix Breitenecker Inge Troch, editor, *Proceedings of the 5th MATHMOD Vienna*, 2006.

[30] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, Martin von Löwis, and Andreas Prinz. Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 7(11):1024–1049, nov 2001.

[31] Joachim Fischer, Michael Piefel, and Markus Scheidgen. A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In Daniel Amyot and Alan W. Williams, editors, *SAM*, volume 3319 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2004.

[32] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.

[33] Rémi Forax, Étienne Duris, and Gilles Roussel. Java Multi-Method Framework. In *TOOLS (37)*, pages 45–56. IEEE Computer Society, 2000.

[34] Martin Fowler. Domain Specific Language. http://martinfowler.com/bliki/DomainSpecificLanguage.html, 2004.

[35] Victoria A. Fromkin and Robert Rodman. *An Introduction to Language*. Holt, Rinehart, and Winston, New York, fourth edition, 1988.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[37] Anna Gerber and Kerry Raymond. MOF to EMF: There and Back Again. In Michael G. Burke, editor, *OOPSLA Workshop on Eclipse Technology eXchange*, pages 60–64. ACM, 2003.

[38] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[39] Ian Graham, Julia Bischof, and Brian Henderson-Sellers. Associations Considered a Bad Thing. *JOOP*, 9(9):41–48, 1997.

[40] T. R. G. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992.

[41] Yuri Gurevich. Evolving Algebras. In *IFIP Congress (1)*, pages 423–427, 1994.

[42] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[43] Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Extending Grammars and Metamodels for Reuse - The Reuseware Approach. *IET Software, Special Issue on Language Engineering*, 2008.

[44] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Oldenbourg R. Verlag GmbH, 3rd edition, 1997.

[45] IBM. SAFARI: A Meta-Tooling Platform for Creating Language-Specific IDEs, 2008.

[46] International Telecommunication Union. SDL 2000, 2000.

[47] Jean Bézivin et al. AMMA. http://www.sciences.univ-nantes.fr/lina/atl/, 2008.

[48] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE*, pages 249–254. ACM, 2006.

[49] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Textual Concrete Syntax (TCS). http://www.eclipse.org/gmt/tcs/, 2008.

[50] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, nov 2014.

[51] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. Wiley-IEEE Computer Society Press, 2008.

[52] Peter Klein and Andy Schürr. Constructing SDEs with the IPSEN Meta Environment. In *Proceedings of the 8th Conference on Software Engineering Environments SEE'97*, pages pp. 2–10, Los Alamitos, Carlifornia, 1997. IEEE Computer Society.

[53] Anneke Kleppe. Towards the Generation of a Text-Based IDE from a Language Metamodel. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007.

[54] Paul Klint, Ralf Lammel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14 (3):331–380, 2005.

[55] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[56] Donald E. Knuth. The Genesis of Attribute Grammars. In Pierre Deransart and Martin Jourdan, editors, *WAGA*, volume 461 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1990.

[57] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006.

[58] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07)*, Montreal, Quebec, Canada, 2007. Technical Report TR-38, Jyväskylä University, Finland.

[59] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 2007.

[60] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Project MontiCore. http://www.sse-tubs.de/monticore/, 2008.

[61] Edward A. Lee. Overview of the Ptolemy Project. University of California, Berkeley, July 2003.

[62] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[63] Markus Völter et al. Open Architecture Ware. http://www.openarchitectureware.org, 2008.

[64] Microsoft. Microsoft Domain-Specific Language (DSL) Tools. http://msdn2.microsoft.com/en-us/library/bb126235.aspx, 2007.

[65] Thomas G. Moher, David C. Mak, Brad Blumenthal, and Laura Marie Leventhal. Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In Cook, Curtis, Scholtz, Jean, and James C. Spohrer, editors, *Empirical Studies of Programmers - Fifth Workshop*, pages pp. 137–161, December 3-15 1993.

[66] Robert N. Moll, Michael A. Arbib, and A. J. Kfoury. *An Introduction to Formal Language Theory*. Springer, 1988.

[67] Marco Mosconi. Durchgängige Modularität in der modellgetriebenen Entwicklung domänenspezifischer Modellierungssprachen mit Hilfe aspektorientierter Programmierung. In Thomas Kühne and Wolfgang Reisig, editors, *Modellierung 2008*, pages 233–237. Gesellschaft für Informatik, März 2008.

[68] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[69] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-Driven Analysis and Synthesis of Concrete Syntax. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 98–110. Springer, 2006.

[70] netbeans. Meta Data Repository. http://netbeans, 2008.

[71] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The Fujaba Environment. *in Proc. ICSE 2000 - The 22nd International Conference on Software Engineering*, pages 742–745, June 2000.

[72] Object Management Group. Interface Definition Language, Version 2.0. http://www.omg.org/gettingstarted/omg_idl.htm, 2003.

[73] Object Management Group. UML 2.0 Infrastructure, 2004.

[74] Object Management Group. UML 2.0 Superstructure, 2004.

[75] Object Management Group. *SAS(R) 9.1.3 Java Metadata Interface: User's Guide*. SAS Institute, Incorporated, 2004.

[76] Object Management Group. OCL 2.0 Specification, June 2005. ptc/2005-06-06.

[77] Object Management Group (OMG). Meta Object Facility (MOF) Specification 1.x. formal/2002-04-03, April 2002.

[78] Object Management Group (OMG). Meta Object Facility (MOF) Specification 2.0 MOF 2.0 to OMG IDL Mapping. ad/04-01-16, January 2004.

[79] Object Management Group (OMG). Meta Object Facility (MOF) Specification 2.0 Core. formal/2006-01-01, April 2006.

[80] Object Management Group (OMG). XML Meta-Data Interchange XMI 2.1.1. formal/2007-12-01, 2007.

[81] Object Management Group (OMG). Meta Object Facility (MOF) Specification 2.0 Facility and Object Lifecycle. ptc/2008-02-20, February 2008.

[82] Object Web – Open Source Middleware. ModFact (Model Factory). http://forge.objectweb.org/projects/modfact/, 2008.

[83] Open Architecture Ware. xText. http://www.openarchitectureware.org/, 2007.

[84] Marian Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Commun. ACM*, 38(6):33–44, 1995.

[85] Gordon D. Plotkin. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[86] PragmaDev. Real Time Developer Studio. http://www.pragmadev.com/, 2008.

[87] Andreas Prinz, Markus Scheidgen, and Merete Skjelten Tveit. A Model-Based Standard for SDL. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL Forum*, volume 4745 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[88] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Scie. Springer-Verlag, Berlin, Germany, 1985.

[89] Steven P. Reiss. PECAN: Program Development Systems that Support Multiple Views. *IEEE Trans. Softw. Eng.*, 11(3):276–285, 1985.

[90] Fritz Ritzberger. Runtime Compiler Compiler. http://runcc.sourceforge.net/, 2004.

[91] Jim R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, 1st edition, October 1990.

[92] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev1, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. LINA, Université de Nantes, 2006. Research Report 06.02.

[93] Daniel A. Sadilek. Energy-aware compilation for Wireless Sensor Networks. In *MidSens '07: Proceedings of the International Workshop on Middleware for Sensor Networks*, Newport Beach, USA, November 2007. ACM Press.

[94] Daniel A. Sadilek. Prototyping and Simulating Domain-Specific Languages for Wireless Sensor Networks. Informatikbericht 217, Humboldt-Universität zu Berlin, Berlin, Germany, November 2007.

[95] Daniel A. Sadilek. Prototyping Domain-Specific Languages for Wireless Sensor Networks. In Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter, editors, *ATEM 07: 4th International Workshop on Software Language Engineering*, number 4 in Mainzer Informatik-Berichte, pages 76–90. Johannes Gutenberg Universität Mainz, October 2007.

[96] Daniel A Sadilek. ESemantics. http://esemantics.sourceforge.net, 2008.

[97] Daniel A Sadilek and Guido Wachsmuth. EPromote. http://epromote.sourceforge.net, 2008.

[98] Daniel A Sadilek and Guido Wachsmuth. EProvide. http://eprovide.sourceforge.net, 2008.

[99] Daniel A. Sadilek and Guido Wachsmuth. EProvide 2.0: an Extensible Framework for Describing Operational Semantics. Technical report, Humboldt-Universität zu Berlin, 2008.

[100] Daniel A. Sadilek and Guido Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA 2008*, volume 5095 of *LNCS*, pages 63–78, Berlin, Germany, 2008. Springer.

[101] Markus Scheidgen. *Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000.* Humboldt Universität zu Berlin, June 2004. master thesis.

[102] Markus Scheidgen. CMOF-Model Semantics and Language Mapping for MOF 2.0 Implementations. In *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 84–93, Washington, DC, USA, 2006. IEEE Computer Society.

[103] Markus Scheidgen. Model Patterns for Model Transformations in Model Driven Development. In Ricardo J. Machado, Joao M. Fernandes, Matthias Riebisch, and Bernhard Schätz, editors, *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 149–158, Los Alamitos, Carlifornia, 2006. IEEE Computer Society.

[104] Markus Scheidgen. Textual Editing Framework. http://tef.berlios.de, 2007.

[105] Markus Scheidgen. A MOF 2 for Java and MOF Action Semantics (MAS). http://amof2.berlios.de, 2008.

[106] Markus Scheidgen. Integrating Content Assist into Textual Modelling Editors. In *Modellierung*, pages 121–131, 2008.

[107] Markus Scheidgen. Textual Modelling Embedded into Graphical Modelling. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA 2008*, pages 153–168. Springer, 2008.

[108] Markus Scheidgen and Joachim Fischer. Human Comprehensible and Machine Processable Specifications of Operational Semantics. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2007.

[109] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[110] Charles Simonyi. Intentional Programming Overview. http://research.microsoft.com/ip/overview/, 1996.

[111] Michael Soden and Hajo Eichler. An Approach to use Executable Models for Testing. In Manfred Reichert, Stefan Strecker, and Klaus Turowski, editors, *EMISA*, volume P-119 of *LNI*, pages 75–85. GI, 2007.

[112] Michael Soden and Hajo Eichler. M3 Actions. http://m3action.sourceforge.net, 2008.

[113] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1977.

[114] Harald Störrle and Jan Hendrik Hausmann. Towards a Formal Semantics of UML 2.0 Activities, 2005.

[115] Team kermeta. kermeta. http://www.kermeta.org/, 2008.

[116] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981.

[117] The Eclipse Foundation. Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf/, 2007.

[118] The Eclipse Foundation. Eclipse Graphical Editing Framework (GEF). http://www.eclipse.org/gef/, 2007.

[119] The Eclipse Foundation. Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/, 2007.

[120] The Eclipse Foundation. Eclipse Modeling MDT. http//www.eclipse.org/modeling/mdt, 2008.

[121] Juha-Pekka Tolvanen. MetaEdit+: Integrated Modeling and Metamodeling Environment for Domain-Specific Languages. In Peri L. Tarr and William R. Cook, editors, *OOPSLA Companion*, pages 690–691. ACM, 2006.

[122] University of California, Berkeley. Ptolemy Project. http://ptolemy.eecs.berkeley.edu/, 2008.

[123] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M.and T. Kuipers de Jonge, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF

Meta-Environment: A Component-Based Language Development Environment. In R. Wilhelm, editor, *Proc. Int'l Conf. Compiler Construction (CC)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[124] Hans Vangheluwe and Juan de Lara. XML-Based Modeling and Simulation: Meta-Models are Models too. In Jane L. Snowdon and John M. Charnes, editors, *Winter Simulation Conference*, pages 597–605. ACM, 2002.

[125] Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 2005.

[126] Alanna Zito, Zinovy Diskin, and Jürgen Dingel. Package Merge in UML 2: Practice vs. Theory? In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2006.

Index

List of Figures

List of Tables